TU WIEN Informatics

Querying Knowledge Graphs at Web Scale

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Amr Azzam, MSc.
Registration Number 11911082

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dr. Axel Polleres

The dissertation has been reviewed by:

Reviewer 1	Reviewer 2

Vienna, 10th May, 2023

Amr Azzam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Amr Azzam, MSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Mai 2023

Amr Azzam

Acknowledgements

Abstract

While Linked Data (LD) provides standards for publishing (RDF) and (SPARQL) querying Knowledge Graphs (KGs) on the Web, serving, accessing and processing such open, decentralized KGs is often practically impossible, as query timeouts on publicly available SPARQL endpoints show. To this end, Linked Data Fragments (LDF) have introduced a foundational framework that has sparked research exploring a spectrum of potential Web querying interfaces between server-side query processing via SPARQL endpoints and client-side query processing of data dumps. Current proposals in between typically suffer from an imbalanced load on either the client or the server. In this thesis, we present a novel approach to share the load between servers and clients, while significantly reducing data transfer volume, by combining server-side query processing with shipping compressed KG partitions. Next, we present the first work that combines both client-side and server-side query optimization techniques in a truly dynamic fashion by employing a cost model that dynamically delegates the load between servers and clients by combining client-side processing of shipped partitions with efficient server-side processing of star-shaped sub-queries, based on current server workload and client capabilities. Thereafter, we investigate alternative interfaces able to ship partitions of KGs from the server to the client, aiming to reduce server-resource consumption. To this end, we align formal definitions and notations of the original LDF framework to uniformly present partition-based LDF approaches. Our thesis is a step forward towards a better-balanced share of the query processing load between clients and servers by shipping graph partitions driven by the structure of RDF graphs to group entities described with the same sets of properties and classes. Throughout the thesis, we empirically evaluate our approach against real-world and synthetic RDF KGs on both pre-existing benchmarks for highly concurrent query execution as well as a novel query workload benchmark inspired by query logs of existing SPARQL endpoints. Our experiments show that our proposed work significantly outperforms state-of-the-art solutions in terms of average total query execution time per client, while at the same time decreasing network traffic and increasing server-side availability and outperforms state-of-the-art solutions and increasing server-side availability towards more cost-effective and balanced hosting of open and decentralized KGs.

Contents

Abstract **vii**

Contents **ix**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Challenges . . . 4
1.4 Hypotheses and Research Questions . . . 4
1.5 Contributions . . . 6
1.6 Thesis Structure . . . 9
1.7 Impact . . . 10

2 Background **13**
2.1 The Semantic Web . . . 13
2.2 The Resource Description Framework . . . 14
2.3 From Linked Data to a network of interconnected KGs . . . 18
2.4 Knowledge Graphs . . . 21
2.5 The SPARQL Query Language . . . 22
2.6 HDT . . . 33

3 A uniform characterization of existing Web querying interfaces **39**
3.1 Linked Data Fragments framework . . . 39
3.2 Partition-based LDF . . . 47

4 Hybrid Shipping for SPARQL Querying on the Web **49**
4.1 Family-Based Partitioning of RDF Graphs . . . 50
4.2 SMART-KG: Design and Overview . . . 53
4.3 Proof of `smart-KG` Correctness . . . 64
4.4 `smart-KG` as an LDF interface (SKG) . . . 71
4.5 Experimental Evaluation . . . 72
4.6 Summary and Limitations . . . 88

5 A Balanced Access to Web Knowledge Graphs **89**

5.1 Motivating Example 90
5.2 WiseKG 92
5.3 Query Processing 96
5.4 Experimental Evaluation 99
5.5 Summary and Limitations 109

6 Smart-KG$^+$: Further Optimizations of Family-partition-based LDF 113
6.1 Partition-based Linked Data Fragments: Typed-Family Partitioning . 114
6.2 SMART-KG$^+$: Design and Overview 117
6.3 SMART-KG$^+$ Extending Partition Generator 118
6.4 SMART-KG$^+$: Query Processing 119
6.5 Experimental Evaluation 132
6.6 Lesson Learned 160
6.7 Summary and Limitations 161

7 Partition-based Linked Data Fragments: Alternatives 163
7.1 Vertical Partitioning (VP) 163
7.2 Horizontal/Range/Sharding Partitioning 164
7.3 Hash Partitioning (HP) 166
7.4 Workload-aware partitioning 166
7.5 K-way Partitioning (KP) 167

8 Reproducibility 169
8.1 Linked Data Fragments Implementation 170
8.2 Deploying Our Experiments 176
8.3 Comunica Implementation 180

9 Conclusion 183
9.1 Summary of Contributions 183
9.2 Critical Assessment of Research Questions 186
9.3 Open Challenges and Future Research Directions 188

List of Figures 197

List of Tables 199

List of Algorithms 201

Bibliography 203

CHAPTER 1

Introduction

> Share your knowledge. It is a way to achieve immortality.
>
> Dalai Lama

1.1 Motivation

The proliferation of the World Wide Web [BF00a] has greatly facilitated the dissemination of knowledge [FLM98] by enabling individuals to publish and access a vast amount of information on various subjects through human-readable HTML documents [LLWL08], forming the "Web of Documents". In recent years, the content on the Web has evolved from a network of linked documents to a more interactive platform containing various forms of human-oriented content such as text, images, sounds, and videos.

Despite the abundance of resources, humans struggle with performing tasks that require accurate and efficient searching, processing, and retrieving of a large amount of data. On the other hand, machines can automate increasingly complex tasks such as *processing queries* over data published on the Web. However, most Web content is somewhat enigmatic to machines due to the human-centered nature of the Web. This has led to the emerging vision of the "Web of Data", an evolution of the Web that includes a set of practices and standards for publishing content on the Web in machine-readable formats by which machines can collaborate on a wide range of tasks and exchange and process data on the Web [BF00b].

Along the lines of this vision, Berners-Lee [BL98] introduced the Semantic Web [BLHL01], which aims to create a Web that is more accessible to machines, enabling software agents

to perform tasks without human intervention. In order to promote the adoption of respective standards, the Semantic Web community [BHB09] has proposed the use of Linked Data principles [BL06, HB11], which involve publishing and linking data in a manner that allows the Web to be queried like "one giant database" [BF00b]. The key technologies of Linked Data are the Resource Description Framework (RDF) and SPARQL, the standard query language for RDF graphs. Driven by the Linked Open Data Initiative (LOD), many openly accessible RDF graphs in various domains such as Life Sciences, Media, Geography, and Linguistics have been published according to Linked Data principles, forming interlinked open knowledge graphs (KGs) such as DBpedia[ABK+07] and Wikidata [VK14], which jointly contribute to a Web of Data [BF00b].

SPARQL queries over open knowledge graphs on the Web are typically executed through the use of clients (issuing SPARQL queries) and servers (exposing RDF graphs via a SPARQL interface). Based on the terminology used in distributed database systems, query workload distribution between clients and servers can be classified into three main types of *shipping strategies* [FJK96]: (i) *Query shipping* in which query execution is completed on the server and only the results are shipped back to the client; (ii) *Data shipping* exploits the processing power of clients (in the extreme case, meaning to simply serve dataset dumps for download) and thereby reduce the workload on servers, and (iii) *hybrid shipping* in which queries are decomposed into subqueries and the processing load is distributed between clients and servers.

Data publishers have so far primarily offered query and data shipping to consume RDF knowledge graphs (KGs). Most open RDF KGs, such as DBpedia and Wikidata, are accessible through a SPARQL endpoint, which is a server that can evaluate SPARQL queries sent by clients over the network. While SPARQL endpoints can perform well for single queries, they can be expensive to host and maintain when serving large KGs or allowing concurrent execution of complex queries by multiple users [AHUV13, SHA+12]. As an alternative, data shipping allows data publishers to provide simple file access to the entire RDF KG and enables clients to download and locally query full data dumps. While data shipping has the advantage of reducing the workload on servers, it can be costly for clients in terms of bandwidth and processing power; plus, it may not be suitable for fast evolving/changing Knowledge Graphs due to the overhead of reshipping the entire graph.

In this context, this thesis aims to explore and optimize the use of *hybrid* shipping for SPARQL query processing over open KGs on the Web: the main goal is to provide a Web querying interface that can effectively balance the workload between clients and servers while minimizing the overhead of communication and data transfer.

1.2 Problem Statement

The term *Knowledge Graph* refers to a scalable data management concept that represents facts about entities and their relationships [BDPP18] in a graph structure. While the term Knowledge Graph points to several technologies and standards as we will detail in Section 2.4, in this work, we specifically focus on *Open* RDF Knowledge Graphs

that are published on the web and adhere to Linked Data principles using the Resource Description Framework (RDF) as the graph-based representation model and the SPARQL query language to retrieve and manipulate these Knowledge Graphs. In this thesis, we address *query processing* over such Knowledge Graphs, which refers to the process of evaluating SPARQL queries over RDF graphs. Various architectures can be used for evaluating SPARQL queries over RDF Knowledge Graphs, such as federated [APU14], peer-to-peer [CF04], and client-server [VSH+16]. Herein, we focus on SPARQL query processing[1] over RDF graphs in a *client-server architecture*.

In a client-server environment, processing SPARQL queries using query or data shipping strategies can be inefficient, particularly when handling high levels of concurrent querying of large RDF graphs [GGvHS10] such as DBpedia. This has contributed to the problem of servers availability [VSH+16]. According to SPARQLES [VUM+17], a service that monitors 565 SPARQL endpoints, 64% of them were unavailable (as of December 2022). In an effort to address this bottleneck, the Linked Data Fragments (LDF) framework [VSH+16] has been proposed as a foundational framework to decentralize query processing tasks to alleviate the burden on data providers hosting such servers. The LDF framework has sparked research on *hybrid shipping* strategies, exploring a spectrum of potential Web querying interfaces in between query shipping, i.e., a full server-side query processing via SPARQL endpoints and data shipping, i.e., full client-side query processing via shipping an entire RDF graph data dump with the aim of finding a balanced client-server load distribution.

Triple Pattern Fragment (TPF) [VSH+16] is an early LDF implementation that enables efficient SPARQL querying by limiting the capabilities of servers to simple triple pattern lookups and transferring the processing of complex patterns to the client side. However, this approach can lead to a decrease in query performance and an increase in network traffic due to the transfer of intermediate results. Bindings-Restricted Triple Pattern Fragments (brTPF) [HA16] attempts to improve the performance of TPF by distributing the join operations between the client and the server. While this reduces the number of HTTP requests and the amount of data received compared to TPF, the number of requests remains relatively high, and intermediate results must still be transferred verbatim. SaGe [MSM19] is a SPARQL query engine designed to address the issue of simple queries being starved of resources due to the execution of long-running server-side queries by introducing a preemptive execution model and scheduling mechanism that allocates a fixed amount of time, called a quantum, for each query and improves average workload completion time by suspending and resuming queries from different clients. However, SaGe may still have issues with high numbers of requests, query context switching, and client-side overhead.

In summary, many web interfaces proposed in the literature so far, including TPF [VSH+16], brTPF [HA16], and SaGe [MSM19], still suffer from an imbalanced load on either the client-side (dumps and TPF), the server-side (SPARQL endpoints and SaGe), or the

[1]Note that throughout this thesis, we interchangeably use the terms "query processing", "Web querying", and "query evaluation" to describe the process of evaluating SPARQL queries over RDF graphs.

network (TPF and brTPF). To address these issues, we propose a further exploration of Web interfaces based on the LDF framework, which has the potential to provide efficient Knowledge Graph querying on the Web while more evenly distributing the query processing load between servers and clients.

1.3 Challenges

In this thesis, we focus on developing an efficient SPARQL Web querying interface over RDF Knowledge Graphs in a client-server environment. The research problem at hand presents several challenges:

- **C1 Large-scale RDF knowledge graphs:** Maintaining highly available query services for large-scale RDF knowledge graphs on the web can be difficult, especially for those with over a billion triples. Web interfaces such as TPF [VSH+16] and brTPF [HA16] have been developed to enable low-cost hosting of knowledge graphs. However, upon complex queries, serving large-scale knowledge graphs using these interfaces can result in increased network traffic, leading to significant query execution performance degradation [MSM19].
- **C2 Skewed structure of RDF Graphs:** The evaluation of SPARQL queries on RDF graphs can be challenging due to the lack of explicit schema and the skewed structure of many RDF graphs. Skewed graphs [FMPdlFRG18], i.e., graphs with an unbalanced distribution of predicates, are characterized by the presence of "dominant" predicates, which occur with significantly higher frequency compared to other predicates in the graph. As we will discuss, these frequent predicates may negatively impact the query performance of existing approaches.
- **C3 Web environment Dynamicity:** The design and implementation of an efficient SPARQL query service in a web environment is a challenge due to the unpredictable and variable nature of the real-world queries [SAH+15]. Client requests for SPARQL queries can lead to fluctuations in server load, resulting in variable availability of resources such as memory and CPU. These fluctuations, along with the influence of hardware parameters, network delays, and the computational resources of clients, can impact the performance of a query. To address these challenges, it may be necessary to adapt the query execution plan during runtime in order to optimize performance according to this dynamic environment.

1.4 Hypotheses and Research Questions

Our first hypothesis is based on the observation that centralized, clustered, and distributed RDF query processing systems use graph partitioning to improve query performance, system scalability, and query load balancing. We posit that using similar ideas by serving compressed, queryable, and reusable KG partitions that can be shipped, cached,

and locally queried without decompression time on the client-side, we can address the challenge **C1**. By breaking down the knowledge graph into manageable partitions, the partition-shipping based Linked Data Fragments (LDF) interface can efficiently handle queries by fetching and combining relevant data only from the necessary partitions. Consequently, this approach reduces the overall network traffic, thus contributing to improved query execution performance for large-scale RDF knowledge graphs. In light of the defined problem statement in 1.2 and the challenges outlined in Section 1.3, our hypothesis is as follows:

> **Hypothesis 1.** *A partitioning technique that utilizes structural analysis of knowledge graphs (KGs) can be used to develop a novel and effective partition-shipping based LDF interface*

The LDF framework suggests that when querying knowledge graphs on the web, it is necessary to consider the trade-off between expressivity, server availability, and client-side resource consumption. Therefore, we propose the creation of a hybrid LDF interface that combines various LDF interfaces with varying expressivity in order to achieve a better balance between server and client load. Our second hypothesis aims to address both challenges **C1** and **C2**.

> **Hypothesis 2.** *The combination of different Linked Data Fragments (LDF) interfaces with partition shipping can improve server availability and enhance the accumulated query performance of large knowledge graphs (KGs) under concurrent query load with multiple clients.*

By leveraging a combination of different Linked Data Fragments (LDF) interfaces with partition shipping, we can effectively distribute and process specific segments of a query through LDF interfaces that are better suited for the given query and the underlying structure of the knowledge graph (KG). Finally, our second hypothesis aims to address both challenges **C3**:

> **Hypothesis 3.** *Employing a cost model that dynamically picks the best-suited interface per query while taking into account server resources, client capabilities, network bandwidth, structural characteristics of KG, and the characteristics of the query can be used to further optimize the query processing load between client and server.*

Hypothesis 3 directly tackles Challenge C3 as the dynamic nature of the cost model allows for adaptive query execution planning at runtime, ensuring efficient utilization of available resources while handling query fluctuations in the web environment. Starting from the hypothesis above, we formulate the corresponding research question, as follows:

RQ1 *Which strategies leveraging partition shipping can achieve significant speedups to existing Web querying approaches over Knowledge Graphs, and when/better should these strategies be employed?*

Here, specifically, we aim to investigate which *partitioning technique* is suitable for serving complex queries, decomposable into subqueries that can be separately served by such partitions, in a manner that shipping the respective compressed KG partitions to be locally queried on the client-side, can be interleaved in a hybrid strategy with existing Web interfaces.

RQ2 *How can the combination of different LDF interfaces with partition shipping further improve the query performance?*

Here, we aim to investigate and compare different, concrete strategies to dynamically combine our partition shipping technique from **RQ1** with other LDF interfaces to achieve optimal performance, considering different factors such as network bandwidth, as well as server and client resources.

RQ3 *How can we systematically enhance and optimize the partitioning technique to improve performance and scalability in Knowledge Graph querying?*

Here, we aim at exploring refinements of our approach, discussing first whether or how other existing graph partitioning techniques from the literature could be used in our approach, and then how to further engineering refinements could enhance the overall performance of Web querying.

1.5 Contributions

The results of this thesis are contributions to the study and improvement of efficient SPARQL Web querying over large-scale RDF Knowledge Graphs given a high concurrent execution of queries from multiple users. In the following, we detail the contributions of this thesis to the respective research questions *RQ1-RQ3* and hypotheses *H1-H3*:

> **Contribution 1.** *An LDF Web querying interface, named smart-KG [AFA+20], efficiently queries Knowledge Graphs (KGs) on the Web by sharing the load between servers and clients, while significantly reducing data transfer volume by combining TPF with shipping compressed KG partitions.*

The first contribution focuses on investigating both Hypotheses 1 and 2. In the following, we breakdown the first contribution into the following sub-contributions:

- We design a KG partition technique named *family partitioning* designed for large RDF KGs. The server-side maintains compressed and queryable KG partitions that are shipped as intermediate results to be further queried locally on the client-side.

- We combine the Triple Pattern Fragment (TPF) strategy with shipping compressed graph partitions that can be locally queried to increase server availability while achieving competitive performance.
- We introduce client-side query optimization and execution techniques that are able to combine heterogeneous LDF APIs responses, ensuring a correct query evaluation.
- An empirical evaluation of smart-KG on synthetic and real-world KGs and queries, significantly outperforming the state-of-the-art interfaces, providing an efficient SPARQL query processing over large-scale Web KGs while maintaining a cost-effective solution.

As a result of the conducted experiments, we gained insights to validate Hypothesis 1 and Hypothesis 2 and answers to **RQ1**.

> **Contribution 2.** *A novel querying interface, named $WiseKG_{heuristic}$ [AAM+21], dynamically shifts the query processing load between client and server based on a predefined heuristic.*

We detail the second contribution which is a further investigation to validate Hypothesis 2, as follows:

- $WiseKG_{heuristic}$ combines the strengths of two Linked Data Fragments APIs (SPF [AKMH20] and smart-KG [AFA+20]) that enable server-side and client-side processing of star-shaped sub-patterns and further advances them by finding a novel balance between server and client load.
- $WiseKG_{heuristic}$ decides whether subqueries should be processed on the client or on the server based on a predefined heuristic (e.g server CPU usage) to ensure the server availability.

Our conducted experiment is directed to answer **RQ2**. The empirical results show that WiseKG significantly outperforms state-of-the-art stand-alone LDF interfaces on highly demanding workloads, with increasing numbers of concurrent clients, with increasing KG sizes, and on different query shapes.

> **Contribution 3.** *A novel querying interface, named WiseKG [AAM+21], that relies on a dynamic cost model to pick the best-suited API per sub-query based on the current server load, client capabilities, and estimation of necessary data transfer between client and server (for intermediate query results), and network bandwidth.*

In the course of Hypothesis 3, we provide details of our third contribution as follows:

- We study the factors that impact the performance of accessing KGs, including server load, client computing resources, and the size of data transferred over the network
- Based on this study, we replace the heuristic-based approach by employing a dynamic cost model to minimize the total time consumed by client-side and server-side components while considering the current load on the server and the client capabilities.
- *WiseKG*'s cost model improves average query execution time while also reducing resource consumption (including less CPU usage and network traffic) compared to existing interfaces.

Our experimental study findings confirm the superior scalability of WiseKG compared to state-of-the-art systems, answering our research question **RQ2**.

> **Contribution 4.** *A sequence of refinements is proposed to improve query execution time and to reduce the shipped intermediate results including (i) improving family partitioning, (ii) improving client-side query evaluation, and (iii) optimizing join subqueries, (iv) Formal definitions and annotations are aligned with the original Linked Data Fragments (LDF) specifications, and (v) The soundness and completeness of our partition shipping-based query execution approach are rigorously verified.*

The final contribution of this thesis further validates Hypothesis 1 and Hypothesis 2 where we provide an improved partitioning technique, query planning, and query execution, as follows:

- We align the formal definitions and annotations with LDF original specifications to uniformly present different LDF APIs.
- We analyze existing partitioning techniques for RDF graphs, assessing their applicability to serve as partitioning mechanisms for Web querying interfaces.
- We introduce partition-based LDF which generalizes LDF interfaces, and returns compressed and queryable partitions that can be used to answer several triple patterns in a single request.
- We implement a query planner that optimizes join ordering among subqueries of an input query.
- We investigate the suitability of asynchronous iterators for a heterogeneous LDF interface query execution.
- We verify the soundness and completeness of our partition shipping-based query execution approach utilized in smart-KG [AFA+20] and WiseKG [AAM+21].

We present a formalization of existing KG partitioning techniques to answer. In addition, we improve typed family-based partitioning based on query logs insights. Our experiments show that our proposed shipping-based query planner and asynchronous join strategy have improved the query performance, answering **RQ3**.

In summary, with the contributions of this thesis, we consider serious barriers to consuming and using open RDF Knowledge Graphs published on the Web. Throughout this work, we propose a series of interconnected contributions that collectively offer an efficient solution to query Knowledge Graphs (KGs) on the Web balancing the load between servers and clients. In addition, we empirically evaluate the proposed contributions using demanding query workloads on real-world KGs and synthetic KGs up to 1 billion triples.

1.6 Thesis Structure

The remainder of this thesis is structured as follows:

- Chapter 2 (Background) provides an overview of the Semantic Web, Linked Data, and Knowledge Graphs (KGs). We cover the RDF data model, SPARQL query language, and HDT [FMG+13] (a compressed and queryable format for RDF graphs). This background introduces the necessary core concepts and notations used throughout the thesis.

- Chapter 3 (A uniform view of LDF interfaces) investigates state-of-the-art techniques for querying RDF Knowledge Graphs on the Web. We present the Linked Data Fragments (LDF) framework and analyze a variety of different interfaces using this framework. We also align and extend the original LDF notations and definitions to uniformly present these approaches. In addition, we present our novel approach partition-based LDF that ships KG partitions hosted by the server to be locally queried on the client-side to answer a submitted query.

- Chapter 4 (Hybrid Shipping for Web querying) investigates how shipping KG partitions can be used to tackle the problem of availability [VSH+16] while achieving significant speedups to query processing (**RQ1**). We introduce our partition shipping strategy which provides a more balanced client-server load distribution (**Contribution 1**).

- Chapter 5 (A balanced access to Web KGs) investigates strategies for dynamically distributing the query processing load between clients and servers based on the current workload (**RQ2**). We focus on combining the strengths of state-of-the-art LDF interfaces to better balance the workload. We present two strategies for selecting the best-suited Web interface to execute a query: (i) a heuristic-based strategy (**Contribution 2**); and (ii) a cost-based strategy to make efficient use of server resources while maintaining high performance during high load (**Contribution 3**).

- Chapter 6 (Refinements) presents a sequence of refinements and optimizations (**Contribution 4**) to further enhance the performance of Web querying interfaces as well as reduce the network traffic (**RQ3**).

- Chapter 7 (Partition-based LDF: Alternatives) investigates alternative possible implementations of LDF interfaces based on existing partitioning mechanisms (**RQ3**) that can potentially be utilized as a KG partitions shipping strategy from servers to clients, with the aim to further reduce both server-resource consumption and network traffic (**Contribution 4**).

- Chapter 8 (Reproducibility, Extensibility, and Comparability) discusses our strategy for reproducible experiments in Web querying research. We also provide details on the implementation of the family partitioning technique. To ensure the extensibility and comparability of our work, we provide two implementations: (i) an extension of the Java implementation of TPF [VSH+16]; and (ii) an additional client implementation using the Comunica platform [THSV18] to facilitate further research on combining different Linked Data interfaces.

- Chapter 9 (Conclusion and future directions) summarises the thesis results and critically reviews the research questions, hypotheses, and their corresponding contributions. We conclude the thesis with a discussion of the current limitation of our work and we outline promising future research directions.

1.7 Impact

The work presented in this thesis has been published in a number of peer-reviewed international conferences and journals, which we will briefly mention in chronological order.

- An initial research proposal, **Enabling Web-Scale Knowledge Graphs Querying: A Research Proposal** by Amr Azzam, was presented at the **European Semantic Web Conference (ESWC) Doctoral Consortium** in 2020 [Azz20]. In this work, we shape the aim of our research which is to develop a new generation of smart clients and servers to balance the load between servers and clients, with the best possible query execution performance, and at the same time reduce data transfer volume, by combining different linked data interfaces. We define the research problem, literature, hypotheses, contributions, and the evaluation plan that we followed in our experiments.

- Guided by the insights of the research proposal, our paper **SMART-KG: Hybrid Shipping for SPARQL Querying on the Web** by Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres was presented at the **Proceedings of the Web Conference 2020** [AFA+20]– The results are presented in Chapter 4.

- We presented our initial vision of utilizing a server-side cost model to enable query execution over a hybrid LDF interface in our poster paper, **Towards Cost-Model-Based Query Execution over Hybrid Linked Data Fragments Interfaces** by Amr Azzam, Ruben Taelman, and Axel Polleres., which was published at the **European Semantic Web Conference (ESWC) in 2020** [ATP20a]– this work forms the lay out the high-level idea to the approach presented in Chapter 5.

- As a follow-up paper, we presented **WiseKG: Balanced Access to Web Knowledge Graphs** by Amr Azzam, Christian Aebeloe, Gabriela Montoya, Ilkcan Keles, Axel Polleres, and Katja Hose at the **Proceedings of the Web Conference 2021** [AAM+21]– the results are presented in Chapter 5.

- More recently, the work presented in Chapter 6 has been submitted for review; in particular, we have submitted a paper under the title **smart-KG: Partition-Based Linked Data Fragments for Querying Knowledge Graphs** by Amr Azzam, Axel Polleres, Javier D Fernández, and Maribel Acosta to the **Journal of Semantic Web**.

Besides the above papers, the author of the presented thesis has also been involved in other published research works which are partially related to this thesis, while not having contributed directly to its content:

- A position paper, **Towards making distributed RDF processing flinker** by Amr Azzam, Sabrina Kirrane, and Axel Polleres was presented at **4th International Conference on Big Data Innovations and Applications (Innovate-Data 2018)** [AKP18] which proposes to manage the RDF KGs based on Flink stream processing framework [2]. This approach to scalablility process KGs pre-date the solution we then adopted in the presented thesis.

  A workshop paper, **The CitySPIN Platform: A CPSS Environment for City-Wide Infrastructures** by Amr Azzam, Peb Ruswono Aryan, Alessio Cecconi, Claudio Di Ciccio, Fajar J. Ekaputra, Javier David Fernandez Garcia, Sotiris Karampatakis, Elmar Kiesling, Angelika Musil, Pujan Shadlau, Thomas Thurner, and Reka Marta Sabou was presented at the **1st Workshop on Cyber-Physical Social Systems (CPSS) 2019**, co-located with the **9th International Conference on the Internet of Things** [AAC+19]. This work introduces techniques for using knowledge graph technologies to integrate heterogeneous data from semi-structured and unstructured sources, including open data and social data. Our approach was in its beginning developed as a part of this CitySPIN project and with smart city applications in the domain of this applied project in mind.

[2] Flink:https://flink.apache.org/

CHAPTER 2

Background

2.1 The Semantic Web

The Semantic Web [BLHL01] is a framework for creating and using machine-readable data on the World Wide Web (Web) in order to enable more intelligent and automated access to information. It was proposed by Tim Berners-Lee, the inventor of the Web, in 2001 as a way to enhance knowledge dissemination on the traditional Web by publishing information in a format that can be easily understood and processed by computers, as opposed to only being presented in a human-readable format. This vision, known as the "Web of Data," aims to empower software agents to discover, process, and use knowledge automatically, allowing them to perform many tasks currently carried out manually by humans.

To realize this vision, the Semantic Web community advocates for using standards and technologies that make data more accessible and interconnected on the Web. One way in which this is being achieved is through the use of the Resource Description Framework (RDF), a graph-structured data model that was specifically designed to provide flexibility and ease of integration for knowledge from various sources on the Web as we will detail in Section 2.2. By using RDF, the Semantic Web aims to evolve into a single global graph database comprising interconnected documents.

The World Wide Web Consortium (W3C) has proposed SPARQL[1] as a standard querying language for RDF data, which allows users to retrieve and manipulate data stored in RDF format, which we will discuss in detail in Section 2.5. This language allows users to retrieve and manipulate data stored in RDF format. In this thesis, we will focus on RDF data model and SPARQL as key standards within the Semantic Web technology stack that support the creation of a "Web of Data".

[1]SPARQL is a recursive acronym, which stands for SPARQL Protocol and RDF Query Language

2.2 The Resource Description Framework

The Resource Description Framework (RDF) is a W3C-recommended standard for describing and exchanging graph-based data on the Web. As outlined in [SR14], RDF is a graph-based data model that is used to represent information about resources such as documents, people, sensors, etc., and their relationships. The RDF data model organizes this information using a tuple structure called an *RDF triple*. Each RDF triple is composed of a subject, predicate, and object, which are represented in the form of (*Subject*, *Predicate*, *Object*), as follows:

- **Subject**: represents the described resource.
- **Predicate**: describes a relationship or a property that associates the subject with the object.
- **Object**: is a resource or a literal (i.e. a sequence of strings) that is related to the subject.

As illustrated in Figure 2.1, RDF triples can be interpreted as a directed labeled graph structure to represent the relationships between resources. In this representation, the subject and object are depicted as nodes, and the predicate is represented as a directed labeled edge.

The RDF data model provides specific terminology, known as *RDF terms*, to define the different types of nodes that can be used in this graph structure, namely:

- **Uniform Resource Identifier (URI)**: is a standard format to identify any logical or physical resources. We note that RDF specification version 1.1[2] permits Internationalized Resource Identifier (IRI) which is similar to URIs but with expanding the set of permitted characters to include international character sets.
- **Literals**: are atomic data values that represent some piece of information. It can be used to represent simple values[3] such as strings, integers, and booleans, as well as more complex data types such as dates, times, and geographical coordinates. Literals may have an associated data type that specifies the type of value being represented. In an RDF triple, a literal is typically used as the object of the statement, while the subject and predicate are represented using resource identifiers (URIs).
- **Blank node**: (also known as a bnode or anonymous node)[4] is a placeholder used to represent a "anonymous" resource nodes without a globally unique identifier. Blank

[2] https://www.w3.org/TR/rdf11-concepts/
[3] https://www.rfc-editor.org/rfc/bcp/bcp47.txt
[4] https://www.w3.org/wiki/BlankNodes

nodes can be used as the subject or object of an RDF triple and are represented using a special syntax (_:node). Blank nodes are useful for representing resources when the identity of the resource is not relevant or when it is being used to represent an intermediate step in a larger process.

We present the formal definition of *RDF terms* following the notation from Perez et al. [PAG09] and the RDF specification [HPS14], as follows:

Definition 2.1. *(RDF Terms) Let U, B, and L be infinite, mutually disjoint sets of IRIs, blank nodes, and literals, respectively. The set of RDF terms is defined as the union of the aforementioned three sets: $U \cup B \cup L$*

We can construct RDF triples from RDF terms as follows: subjects are admissible to include IRIs or blank nodes, predicates accept only IRIs, and objects can contain any of the RDF terms (IRIs, blank nodes, or literals).

Definition 2.2. *(RDF Triple) An RDF triple $t = (s,p,o) \in (U \cup B) \times U \times (U \cup B \cup L)$ where we refer to the components of a single RDF triple as by $subj(t) = s$ represents the subject of a triple, $pred(t) = p$ represents the predicate, and $obj(t) = o$ represents the object.*

Example 1 presents an RDF triple describing **Ross Geller** who is a fictional character from the TV series **Friends** where the subject is **dbr:Ross_Geller**, the predicate is **rdf:type**, and the object is **dbo:FictionalCharacter**. In RDF, a class represents a concept or category of things that can be described using RDF triples. Classes are used to group together resources that share common characteristics or properties. The rdf:type property, which is predefined in RDF, can be used to specify the class of a resource as Example 1.

Example 1. *(RDF Triple)*

The following is an example of an RDF triple in a form of a tuple using DBpedia ontology [a]:

(dbr:Ross_Geller, rdf:type, dbo:FictionalCharacter)

Subject: dbr:Ross_Geller is an IRI that represents the subject Ross Geller;

Predicate: rdf:type is a property that is used to specify the type or class of a resource. In our example, it states that dbr:Ross_Geller resource is an instance of the class dbo:FictionalCharacter;

Object: dbo:FictionalCharacter is an IRI that represents a class, to which the subject belongs in the DBpedia ontology.

[a]DBpedia Ontology (`http://dbpedia.org/ontology/`) is a vocabulary and ontology developed by the DBpedia community that describes entities and properties in the DBpedia dataset, based on the RDF data model and is expressed using the Web Ontology Language (OWL) [SS04].

As shown in Example 1, prefixes can be used to provide a short form for URIs[5]. For instance, we abbreviate the URI `http://dbpedia.org/resource/Ross_Geller` to **dbr:Ross_Geller** where dbr corresponds to `http://dbpedia.org/resource/`

An RDF graph G is a finite set of RDF triples and is formally defined as follows:

Definition 2.3. *(RDF Knowledge Graph (KG)) An RDF graph G is a subset of $(U \cup B) \times U \times (U \cup B \cup L)$. That is, an RDF graph is a finite set of RDF triples, where subj(G), pred(G), and obj(G) denote subjects, predicates, and objects in G*

Example 2 demonstrates how RDF describes information referring to the fictional character Ross Geller. The example presents 6 facts about the described character in 6 different RDF triples forming an RDF Graph. In Figure 2.1, we show the graphical representation for an RDF graph describing **Ross Geller** character, with IRIs depicted as ovals and literals depicted as rectangles.

Example 2. *(RDF Graph) The following RDF graph describes the resource **Ross Geller** who is a fictional character from the sitcom **Friends** based on DBpedia graph:*

Ross is a fictional character in the TV series Friends.
(dbr:Ross_Geller, rdf:type, dbo:FictionalCharacter)

Ross Geller's character is portrayed by the actor David Schwimmer.
(dbr:Ross_Geller, dbo:portrayer, dbr:David_Schwimmer)

Ross Geller's name is "Ross Geller" in English.
(dbr:Ross_Geller, rdfs:label, "Ross Geller"@en)

Ross Geller was born on 1969-10-18.
(dbr:Ross_Geller, dbo:birthDate, "1969-10-18"^^xsd:date)

[5]In Example 1 and Example 2, we make use of the following prefixes:
dbr:http://dbpedia.org/resource/resource
dbo:http://dbpedia.org/resource/ontology
dbp:http://dbpedia.org/resource/property
rdf:http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:http://www.w3.org/2000/01/rdf-schema#

Ross Geller is a Paleontologist.
(dbr:Ross_Geller, dbo:occupation, dbr:Paleontology).

Ross Geller holds a PhD in Paleontology.
(dbr:Ross_Geller, dbp:title, dbr:Doctor_of_Philosophy).

Ross Geller is married to Rachel Green.
(dbr:Ross_Geller, dbp:spouse, dbr:Rachel_Green).

In practice, RDF graphs can be stored and exchanged using various serialization formats such as RDF/XML [DMvH+00], Notation3 (N3)[BCK+08], JSON-LD[LG12], and Turtle [BB08]. Among these serialization formats, Turtle is notable for its compact and human-readable textual form, making it easy to understand and interpret RDF graphs. Additionally, Turtle includes conventions for representing common patterns of usage, such as abbreviating long URIs, which can reduce the verbosity of the serialization, and representing common data types, such as strings, integers, and floating point numbers. In the following, we serialize the RDF graph that describes **Ross Geller** using Turtle format [BB08]:

Example 3. *This example demonstrates the Turtle serialization of the RDF graph in Example 2:*

```
@prefix dbr: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.
@prefix dbp: <http://dbpedia.org/property/>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

dbr:Ross_Geller   a               dbo:FictionalCharacter;
                  dbo:portrayer   dbr:David_Schwimmer;
                  rdfs:label      "Ross Geller"@en;
                  dbo:birthDate   "1969-10-18";
                  dbo:occupation  dbr:Paleontology;
                  dbp:title       dbr:Doctor_of_Philosophy;
                  dbp:spouse      dbr:Rachel_Green.
```

In Example 3, we use the following three features of the Turtle language:

- **Prefixed names**[6]: These are used to shorten the representations of IRIs by linking a prefix label to the full IRI. In our example, dbo:portrayer is a shortened form of the full IRI `http://dbpedia.org/ontology/portrayer`.
- **Predicate lists**[7]: This feature allows for multiple triples that share the same

[6] `http://www.w3.org/TR/turtle/#iri-a`
[7] `http://www.w3.org/TR/turtle/#predicate-lists`

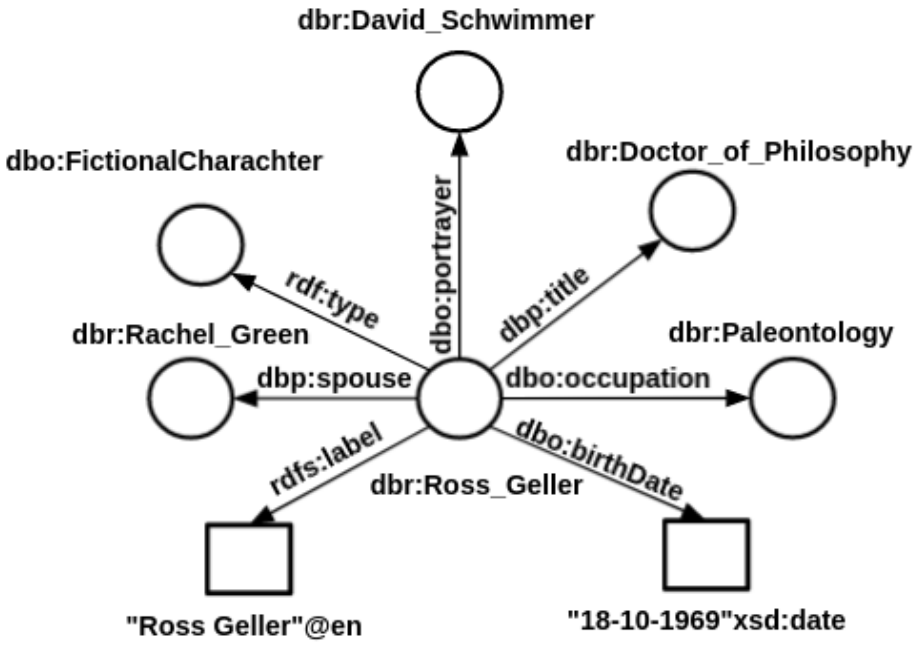


Figure 2.1: A graphical representation of an RDF graph from Examples 2 and 3. The nodes represent subjects and objects and the directed labeled edges represent the predicates.

subject to be grouped together using the ';' symbol. In our example, Ross_Geller is referenced by 7 triples;

- **Token 'a'**: This is used in the predicate position as a shorthand representation of the IRI `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` which is commonly used in RDF data.

2.3 From Linked Data to a network of interconnected KGs

The Semantic Web, through its standards, languages, and protocols, aims to provide a framework for publishing and querying information in a machine-readable format on the Web. However, the challenge of linking and integrating data from diverse Web sources in a coherent and consistent manner remains. To address this issue, in 2006 Tim Berners-Lee introduced the concept of "Linked Data" [BL06, HB11], which outlines a set of principles for publishing and linking data on the Web based on the Semantic Web standards. These principles serve as a conceptual blueprint for creating a Web of Data, where data from various sources is interlinked and accessible through a single global graph.

The design principles for Linked Data, as proposed by Tim Berners-Lee, aim to establish a Web of Data by using Uniform Resource Identifiers (URI) or Internationalized Resource Identifiers (IRI) to identify real-world entities or abstract concepts, and linking those descriptions to other related entities on the Web. An example of how Linked Data can be applied can be demonstrated by "Friends" series, which is identified by its IRI on Wikidata: `https://www.wikidata.org/wiki/Q79784`. Assume that information about Friends is available on various Web sources, each source providing different facts about the show. Through the use of Linked Data principles, a unique identifier, such as an IRI, is assigned to the "Friends" resource, allowing for other Web sources to link to this resource, such as `http://dbpedia.org/resource/Friends`. By dereferencing this IRI via the Hypertext Transfer Protocol (HTTP), we can access structured descriptions of the Friends entities using RDF data model. For instance, the IRI `https://www.wikidata.org/wiki/Q79784` provides information about awards and nominations received by the show, by including links to other related entities such as the Primetime Emmy Awards on the web. The core concept of these principles is to use HTTP-dereferenceable IRIs in the RDF data published on the Web to ease the process of discovering and retrieving more information about related IRIs upon lookup.

In the following, we refer to the Linked Data principles as introduced by Berners-Lee:

- **LDP1** use IRIs as names to refer to *real world* things;
- **LDP2** Use HTTP IRIs so that software agents and people can look up those names (i.e dereferencing the IRIs);
- **LDP3** retrieve useful information upon dereferencing those IRIs using RDF data model and SPARQL query language;
- **LDP4** include links to other externally dereferenceable so that the user could discover more related things.

Linked Data principles delineate how to deploy Semantic Web standards to form a Web of Data. In particular, Linked Data aims at improving the interoperability of data published on the Web with the goal of interlinking this data in such a fashion that the Web can be queried as if it were one "Giant Global Graph" [8].

To put the Linked Data vision into action, the World Wide Web Consortium (W3C) has launched a project called "Linking Open Data" (LOD)[9] to motivate Open Data publishers to create, publish, and interlink their data according to Linked Data principles in the form of open RDF Knowledge Graphs (KGs). Driven by the LOD [BHB09] initiative, the amount of *open* Knowledge Graphs published on the Web has seen continuous growth

[8] `https://web.archive.org/web/20160713021037/http://dig.csail.mit.edu/breadcrumbs/node/215`

[9] `https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`

over the past decade, constructing thousands of interconnected KGs connected as Linked Data, many of which comprise billions of edges [BDPP18]. Examples of such openly available interlinked KGs include DBpedia [LIJ+15, ABK+07], Freebase[10] [BEP+08], Yago [SKW07], and Wikidata [VK14].

However, while publishing open data on the Web according to Linked Data principles has not brought about the full realization of the one single giant graph, the continuous publishing of RDF graphs has rather led to forming a global dataspace of a network of (partially) interconnected open RDF KGs named LOD cloud[11] from diverse knowledge domains including governments (e.g., data.gov[12], Vienna History Wiki[13]), media (e.g BBC[14] [KSR+09], New York Times[15]), life science (e.g., DrugBank[16][WKG+08], PubChem[17][KCC+21]), social networking (e.g, SocialLink[18][NCG18]), and academia (e.g., DBLP[19], UniProt[20]). Figure 2.2 illustrates the most recent version of the LOD cloud where each node represents an RDF graph and each edge is a link between two different graphs. As of October 2022, the LOD cloud contains 1,255 datasets[21].

These open KGs are typically queryable via SPARQL Web querying interface, downloadable data dumps which involve shipping massive data over the network, or by just "following your nose", i.e., gathering data about entities via their URIs dereferenceable as HTTP links [Har13]. We detail the various techniques of Web querying of open KGs in Chapter 3

Despite a decade of research on Linked Data, there are still significant barriers to providing stable and responsive query services for open KGs published in RDF. One main issue is the service availability of public SPARQL query interfaces [VSH+16], i.e the capability to handle requests in an efficient and timely manner. This service availability issue has a direct impact on achieving the vision of seamless live querying of KGs as the constitutes of the Web of Data which is the main focus of this thesis.

[10] In Fact, freebase, after being one of the first openly available KGs, has been discontinued and commercially been acquired and subsumed in Google's KG, cf. `https://developers.google.com/freebase`, last accessed 23/04/2021

[11] `https://lod-cloud.net/`

[12] `https://data.gov/`

[13] `https://www.geschichtewiki.wien.gv.at/RDF`

[14] `https://www.bbc.co.uk/ontologies`

[15] `https://developer.nytimes.com/`

[16] `https://go.drugbank.com/`

[17] `https://pubchem.ncbi.nlm.nih.gov/`

[18] `http://sociallink.futuro.media/`

[19] `https://dblp.org/rdf/`

[20] `https://www.ebi.ac.uk/rdf/`

[21] The RDF datasets are connected through 16,174 dataset pairs that have one or more links between their members. For a more detailed analysis of the links between knowledge graphs on the web, refer to [HFKP20]

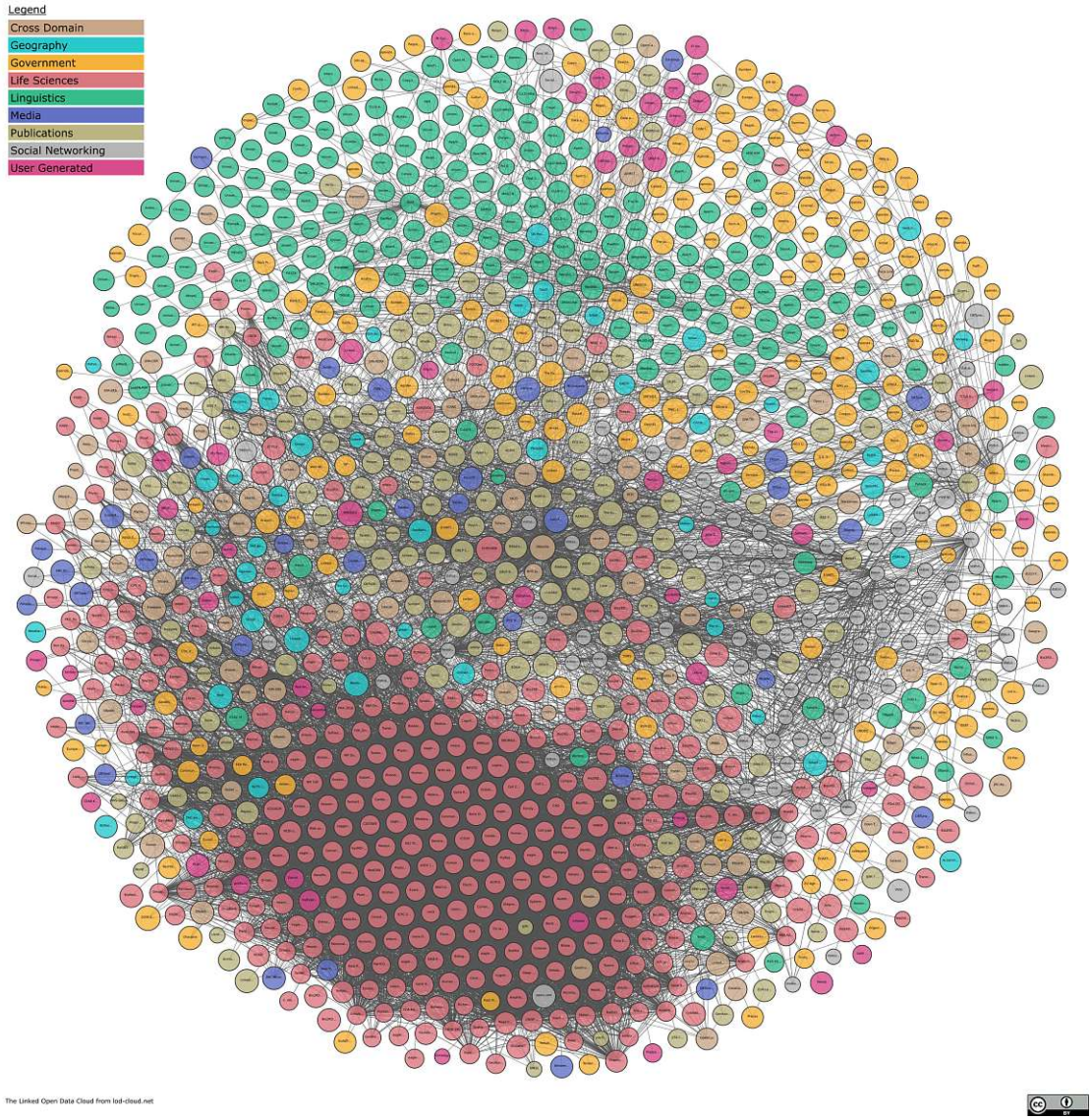


Figure 2.2: The LOD cloud diagram, as of October 2022

2.4 Knowledge Graphs

Independent of the Semantic Web and Linked Data alone, Knowledge Graphs (KGs) have emerged as a promising data management foundation to provide scalable knowledge models that represent a collection of interlinked, diverse, and heterogeneous facts about entities and the relations among these diverse entities [BDPP18].

The concept of Knowledge Graphs, which utilizes a wide range of technologies such as RDF and often graph data platforms, has gained significant attention in recent years. In 2012, Google introduced its own Knowledge Graph to enhance the results of its search

engine service. Both Google's Knowledge Graph and RDF-based knowledge graphs use a combination of entities (such as people, places, and things) and the relationships between them to organize and present information. It is worth noting that Google Knowledge Graph API generates responses presented in JSON-LD and compatible with schema.org schemas, which is derived from RDF schema.

The KG concept has since been adopted by a variety of companies to create new value in terms of commercial applications, such as Google Search[22], Amazon Alexa[23], and Bloomberg [ZMBR20]. Additionally, various graph data platforms, such as Neo4j[24], Redis[25], and GraphQL[26], have also been implemented to support the development and deployment of Knowledge Graphs, independent and orthogonal to standard RDF and SPARQL technologies.

In addition to these efforts, specific domains have recognized the potential of Knowledge Graphs for scalable data integration through the provision and interlinking of diverse datasets in fields such as cancer research [HRW+20, Kam19], and drug discovery [HKH+14]. For instance, biomedical research attempts to construct and interlink medical KGs [ESW15, MPS+17] by harvesting heterogeneous data sources such as research papers, patents, clinical trials, and patient records. This forms newly inferred relationships between biological entities such as genes, symptoms, and diseases assessing the drug discovery process [HKH+14, KFP+19].

The concept of Knowledge Graphs can be connected to the Web of data by considering open Knowledge Graphs like DBpedia as examples of the Web of data. Meanwhile, Enterprise Knowledge Graphs are typically designed for internal use in organizations and are not publicly accessible, but can still be viewed as valuable external resources for the Web of Data. In a broader sense, the Web of Data can be visualized as a collection of loosely connected Knowledge Graphs.

While, in practice, there are several notations and technologies that adopt the Knowledge Graph concept, in this thesis, we focus on *open RDF Knowledge Graphs*.

2.5 The SPARQL Query Language

The SPARQL Protocol and RDF Query Language (SPARQL) is W3C's recommended language to retrieve and manipulate RDF data. SPARQL is a declarative language where queries are constructed following graph-based templates that are matched against the RDF graph, following an approach known as graph pattern matching. By means of examples, we introduce the core features supported by the language required throughout this thesis.

[22] https://cloud.google.com/enterprise-knowledge-graph/docs
[23] https://www.aboutamazon.com/news/devices/how-alexa-keeps-getting-smarter
[24] https://neo4j.com/
[25] https://redis.com/
[26] https://graphql.org/

The core query atom of SPARQL is a *triple pattern*. Triple patterns extend RDF triple, where apart from RDF terms variables are also permitted in subject, predicate, object positions. Variables can substitute any RDF term to serve as placeholders that are bounded to an RDF term in the query solution.

The definition of a triple pattern [27] is as follows:

Definition 2.4. *(Triple Pattern) Let V be the set of variables, disjoint from the set of RDF terms such that $V \cap (U \cup B \cup L) = \phi$. A triple pattern tp is defined as $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$, where the components subject, predicate, and object can represent an RDF term or a variable.*

Example 4. *(Triple Pattern)*

Variables are distinguished from RDF terms with the '?' prefix and can be used in triple patterns as placeholders for RDF terms[a]. For example, the triple pattern:

?character dbo:portrayer dbr:David_Schwimmer.

can be used to query an RDF graph to retrieve the IRIs of the characters that David Schwimmer has portrayed. Given an RDF graph, this triple pattern would match any triples in the graph with a predicate of dbo:portrayer and an object of dbr:David_Schwimmer, and bind the subject to the variable ?character. For example, given the RDF graph in Example 2, the variable ?character can be mapped to dbr:Ross_Geller.

[a]The syntax of a SPARQL query is inspired by Turtle syntax in permitting the definition of prefixes and base IRI.

Basic Graph Patterns (BGP) are sets of triple patterns, that can also be understood as conjunctive queries over an RDF graph. The definition of a basic graph pattern is as follows:

Definition 2.5. *(Basic Graph Pattern)*

Let $tp_1, tp_2, \ldots, tp_n$ be triple patterns. A basic graph pattern is a set of triple patterns and is often represented as a conjunctive of triple patterns where $\cdot$ represents the logical conjunction operator.

[27]Note that Without loss of generality we do not consider blank nodes in triple patterns, as these We do not consider blank nodes in patterns as these can be semantically equivalently replaced by variables [dBFT05]

$$Q = \{tp_1, tp_2, \ldots, tp_n\}$$

Triple patterns can be combined using various expressions, such as filtering (Filter), conjunction (.), disjunction (Union), and optional patterns (Opt), to form more complex graph patterns known as SPARQL expressions. The SPARQL specification also defines different query forms:

- Ask: returns a Boolean indicating whether a given SPARQL graph pattern can be matched (True) or not matched (False) to a graph;
- Construct: retrieves an RDF graph that is formed by combining triples generated from replacing variables of a graph pattern by their solutions;
- Select: retrieves the solutions to the variables in a given SPARQL graph pattern.

According to SPARQL [GHMP11], a SPARQL graph pattern is defined as follows:

Definition 2.6. *(SPARQL Graph Pattern) A SPARQL graph pattern is recursively defined*

- *a triple pattern tp is a SPARQL graph pattern;*
- *if Q_1 and Q_2 are graph patterns,*
  then Q_1 . Q_2 (conjunctive expression),
  Q_1 Opt Q_2 (optional expression), and
  Q_1 Union Q_2 (union expression) are graph patterns;
- *if Q_1 is a graph pattern and R is a Filter condition, then Q_1 Filter R is a graph pattern (filter expression).*

In this thesis, our main focus is on the SPARQL Select query type which returns a set of bound variables which we define as follows:

Definition 2.7. *(Select Query) Let Q be a graph pattern and we denote by $S \subseteq var(Q)$ is a finite set of the projected variables of Q. A SPARQL Select query is a graph pattern of the form $\text{Select}_S(Q)$*

In the following, we provide an example of a full SPARQL query to retrieve the results of the triple pattern example 4:

Example 5. *(Simple Triple Pattern Query) The following SPARQL query is of the type* SELECT *query which retrieves the character resources that David Schwimmer has portrayed. The SPARQL query consists of a* SELECT *clause identifying a single variable ?character, which will be bounded to a value that matches the triple pattern defined in the* WHERE *clause:*

@prefix dbr: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.

SELECT ?character
WHERE
{
?character dbo:portrayer dbr:David_Schwimmer.
}

The result of this query based on the RDF Graph in Example 2

?character
http://dbpedia.org/resource/Ross_Geller

In the case of SELECT queries, SPARQL graph patterns are declared within the WHERE clause to be matched with the RDF graph. The conjunction of graph patterns (And) is written as a period (·). Example 6 describes how a basic graph pattern can be utilized to match a subgraph:

Example 6. *(Basic Graph Pattern) The following SPARQL query consists of a basic graph pattern (BGP) of three triple patterns. The BGP retrieves all the variables included in the query using the operator (*). The query basically retrieves information about the character who is the spouse of Rachel_Green, his job, and the actor who portrayed it.*

@prefix dbr: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.
@prefix dbp: http://dbpedia.org/property/>.

*SELECT **
WHERE
{
?character dbo:portrayer ?actor.
?character dbo:occupation ?job.

dbr:Rachel_Green dbp:spouse ?character.
}

The result of this query based on the RDF Graph in Example 2

?character	*?job*	*?actor*
dbr:Ross_Geller	*dbr:Paleontology*	*dbr:David_Schwimmer*

The evaluation of SPARQL queries on RDF graphs involves the use of mappings, which instantiate variables in a SPARQL graph pattern with RDF terms. Each solution mapping represents an answer for a given SPARQL graph pattern.

Definition 2.8. *(SPARQL Solution Mappings) For any a BGP Q, we denote by $var(Q)$ its variables. The solutions of a (query) pattern Q over a graph G, denoted $[\![Q]\!]_G$, are given as a set Ω of bindings, i.e. mappings of the form, $\omega : var(Q) \rightarrow (U \cup B \cup L)$, such that $G \models \omega(Q)$, i.e. $\omega(Q)$ forms a (sub)graph entailed by G. Two mappings ω_1, ω_2 are called compatible, denoted as $\omega_1 \| \omega_2$ if for any $v \in dom(\omega_1) \cap dom(\omega_2)$, $\omega_1(v) = \omega_2(v)$ where the domain of a mapping ω, $dom(\omega)$, is the subset of V for which ω is defined.*

The solution of a SPARQL graph pattern or SPARQL query can be defined by the following algebra:

Definition 2.9. *(SPARQL Algebra) Let Ω, Ω_1, Ω_2 be SPARQL bindings sets, R is a filter condition, and S denotes a finite set of variables where $S \subset V$. We define the SPARQL algebra operators (join ($\bowtie$), union(σ), minus ($\setminus$), left outer join (⟕), projection (π), and selection (σ)), as follows:*

$\Omega_1 \bowtie \Omega_2 := \{\omega_1 \cup \omega_2 \mid \omega_1 \in \Omega_1, \omega_2 \in \Omega_2 : \omega_1 \parallel \omega_2\}$

$\Omega_1 \cup \Omega_2 := \{\omega_1 \mid \omega_1 \in \Omega_1 \vee \omega_1 \in \Omega_2\}$

$\Omega_1 \setminus \Omega_2 := \{\omega_1 \in \Omega_1 \mid \forall \omega_2 \in \Omega_2 : \omega_1 \nparallel \omega_2\}$

$\omega_1 \text{⟕} \Omega_2 := \{\Omega_1 \bowtie \Omega_2\} \cup \{\Omega_1 \setminus \Omega_2\}$

$\pi_S(\Omega) := \{\omega_1 | \exists \omega_2 : \omega_1 \cup \omega_2 \in \Omega \wedge dom(\omega_2) \subseteq S \wedge dom(\omega_2) \cap S = \phi\}$

$\sigma_R(\Omega) := \{\omega \in \Omega \mid \omega \models R\}$, *where $\models$ tests if a mapping ω satisfies the applied filter condition R.*

The evaluation of any SPARQL query Q over an RDF Knowledge Graph G requires a function that defines the evaluation semantics of the SPARQL query and its graph patterns. Following Perez et al. [PAG09] and Schmidt et al. [SML10], we denote the

evaluation function as $[\![\cdot]\!]_G$, which maps SPARQL patterns from Definition 2.6 into SPARQL set Algebra, as follows:

Definition 2.10. *(SPARQL Set Semantics) Let G be an RDF graph, tp a triple pattern, Q, Q_1, Q_2 SPARQL graph patterns, a filter condition R, and $S \subseteq V$ a finite set of variables. The evaluation of the SPARQL graph pattern over G can be defined recursively following SPARQL set semantics:*

$$[\![tp]\!]_G := \{\omega \mid dom(\omega) = vars(tp) \text{ and } \omega(tp) \in G\}$$

$$[\![Q_1 \ . \ Q_2]\!]_G := [\![\Omega_1]\!]_G \bowtie [\![\Omega_2]\!]$$

$$[\![Q_1 \ \textsc{Opt} \ Q_2]\!]_G := [\![\Omega_1]\!]_G \bowtie [\![\Omega_2]\!]_G$$

$$[\![Q_1 \ \textsc{Union} \ Q_2]\!]_G := [\![\Omega_1]\!]_G \cup [\![\Omega_2]\!]_G$$

$$[\![Q_1 \ \textsc{Filter} \ Q_2]\!]_G := \sigma_R([\![Q]\!]_G)$$

$$[\![\textsc{Select}_S(Q)]\!]_G := \pi_S([\![Q]\!]_G)$$

2.5.1 Query Processing and Optimization

In this section, we have presented the semantics of SPARQL. Here, we summarize the query optimization concepts required in this thesis.

The query optimization problem can be formally defined as a search problem [Ioa96] in which the objective is to identify an optimal evaluation plan from the space of possible plans that results in efficient query execution. The optimization process involves traversing the search space of plans by comparing equivalent plans, defined as plans that produce the same results. The space of possible plans is generated based on: (i) the algebraic set of rules that preserve plan equivalence [Ioa96, Cha98], such as join commutativity and associativity, and (ii) the method-structure space [Cha98], which includes the available implementations of logical operations, such as Nested Loop for the join operator. In the context of SPARQL, a nested loop join is an algorithm that combines two sets of bindings using two nested loops, by iterating through one set (the outer loop) and for each triple in that set, iterating through the other set (the inner loop) to find matches. The optimizer utilizes cardinality estimation techniques and a join reordering algorithm to determine the most promising join order while traversing the search space.

Cardinality Estimation is the process of determining the number of (intermediate) results produced by a query or a sub-query, or even an operator. The number of results is an essential factor in determining the cost of executing a query plan. For instance, estimating the intermediate results, or cardinality, of an operator, such as join, is a critical step in determining the overall cost of executing a query plan, as it has a direct impact on the required time and space resources. As a result, an accurate cardinality estimation [NM11, GN14] is essential for query optimizers to provide efficient query plans.

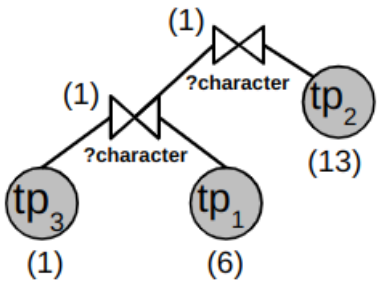


Figure 2.3: A left-linear execution plan based on Nested Loop Join based on a greedy heuristic

Join Reordering refers to the process of rearranging the order of the joins in a query in order to improve the efficiency of the query execution. The optimization process explores the space of possible join reordering plans by implementing a search strategy. The optimizer evaluates the plans generated with the search strategy based on:

- **Greedy Heuristics**: This approach uses heuristics, or rules of thumb, to guide the search for the optimal join order. For example, one heuristic might be to always join the smallest possible number of triples first.
- **Cost Model**: This approach computes a numerical value for each plan based on statistics about the data and the query to estimate the cost of different join orders, and then chooses the order with the lowest estimated cost.

with the objective of identifying the most favorable or, in some cases, optimal plans[28].

Typically, join reordering plans are represented as a binary tree named join trees with join operators as inner nodes and triple patterns as leaf nodes. The joins trees can be of different shapes (left-deep tree, right-deep tree, and bushy tree) and the first three can be referred to as linear trees.

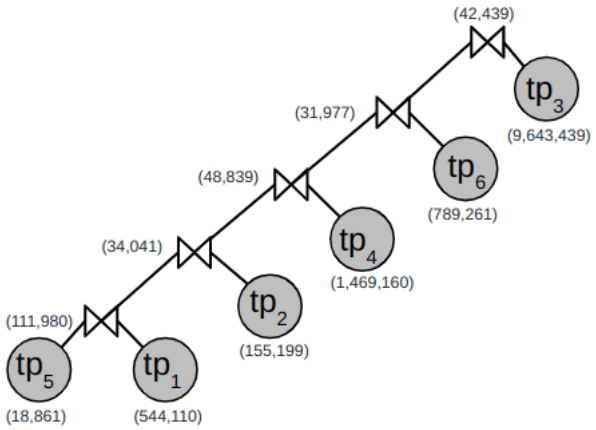
We utilize the query processing and optimization approach for Web querying proposed by Verborgh et al. in [VSH+16] to construct a left-linear deep plan utilizing Nested Loop Joins[29]. As illustrated in Figure 2.3, we demonstrate an execution plan based on this approach to the query examples provided in Example 6 and Example 7 using a greedy heuristic where triple patterns with the least cardinality are joined first. The leaves of the plan represent individual triple patterns and the inner nodes represent Nested Loop

[28] Ibaraki and Kameda [IK84] have formally shown that determining the optimal reordering is an NP-complete problem

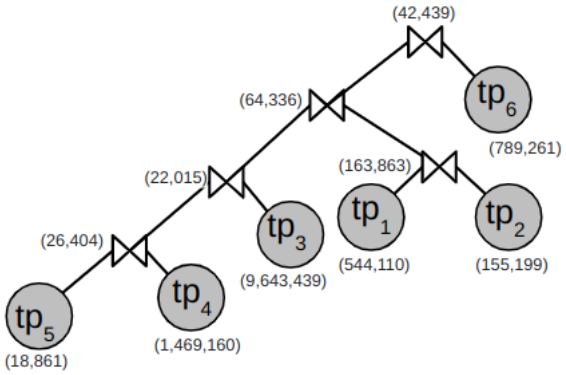
[29] This approach differs from the adaptive Web query processing approach proposed by Acosta and Vidal in [AV15], which utilizes bushy trees and Symmetric Hash Join. However, for the purpose of this thesis, we have chosen to adopt the original implementation from Verborgh et al. in our proposed systems and will investigate the approach proposed by Acosta and Vidal in future work.

```
SELECT * WHERE {
    ?tvprogram dbo:starring ?actress .              # tp1  544,110 matches
    ?tvprogram dbo:releaseDate ?releaseDate .       # tp2  155,199 matches
    ?actress dbo:wikiPageExternalLink ?link .       # tp3  9,643,439 matches
    ?actress dbo:birthPlace ?city .                 # tp4  1,469,160 matches
    ?actress dbp:occupation "Actress"@en.           # tp5  18,861 matches
    ?city dbo:country ?country.                     # tp   789,261 matches
}
```

(a) Select all actresses, their TV programs, and birthplace information



(b) A left-linear plan of the query



(c) Another query plan of the SPARQL query

Figure 2.4: Examples of different query plans of a given SPARQL query

join operators, with the number of matching triples and intermediate results indicated in parentheses.

In Figure 2.4, we present different possible query plans of a more complex query example that we will use in the following chapters. The first query plan is presented in Figure 2.4b which is a left-deep query plan simlar to the query plan divised in Figure 2.3. Figure 2.4c

shows the second query plan and the numbers representing the intermediate results cardinality estimates which we will explain in detail in the rest of the thesis.

In the following, we demonstrate the evaluation of the simple SPARQL query from Example 6 over an extension of the graph described in the Turtle Example 3.

Example 7. *We evaluate the query from Example 6 over the following RDF graph detailed in Turtle format:*

```
@prefix dbr: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.
@prefix dbp: <http://dbpedia.org/property/>.
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

dbr:Ross_Geller      a              dbo:FictionalCharacter;
                     dbo:portrayer  dbr:David_Schwimmer;
                     rdfs:label     "Ross Geller"@en;
                     dbo:birthDate  "1969-10-18";
                     dbo:occupation dbr:Paleontology;
                     dbp:title      dbr:Doctor_of_Philosophy;
                     dbp:spouse     dbr:Rachel_Green;
                     dbp:religion   dbr:Cultural_Judaism;
                     dbp:children   dbr:Ben_Geller.
dbr:Rachel_Green     a              dbo:FictionalCharacter;
                     dbo:portrayer  dbr:Jennifer_Aniston;
                     rdfs:label     "Rachel Green"@en;
                     dbo:occupation dbr:Louis_Vuitton.
dbr:Monica_Geller    a              dbo:FictionalCharacter;
                     dbo:portrayer  dbr:Courteney_Cox;
                     rdfs:label     "Monica Geller"@en;
                     dbo:occupation dbr:Chef;
                     dbp:spouse     dbr:Chandler_Bing;
                     dbo:alias      "Monica Geller-Bing"@en;
                     dbp:religion   dbr:Cultural_Judaism;
                     dbp:children   "Jack and Erica Bing"@en.
dbr:Chandler_Bing    a              dbo:FictionalCharacter;
                     dbo:portrayer  dbr:Matthew_Perry;
                     rdfs:label     "Chandler Bing"@en;
                     dbo:occupation dbr:Statistical_inference.
                     dbp:religion   dbr:Agnosticism;
                     dbp:family     "Charles Bing"@en;
                     dbp:family     "Nora Tyler Bing"@en;
                     dbp:children   "Jack and Erica Bing"@en;
                     dbp:gender     "Male"@en.
dbr:Joey_Tribbiani   a              dbo:FictionalCharacter;
                     dbo:portrayer  dbr:dbr:Matt_LeBlanc;
                     rdfs:label     "Joey_Tribbiani"@en;
                     dbo:occupation "Actor"@en;
                     dbo:alias      "Ken Adams"@en;
                     dbp:religion   dbr:Catholic_Church;
                     dbp:nationality dbr:Italian_Americans;
                     dbp:affiliation dbr:Screen_Actors_Guild.
```

dbr:Phoebe_Buffay	*a*	*dbo:FictionalCharacter;*
	dbo:portrayer	*dbr:Lisa_Kudrow;*
	rdfs:label	*"Phoebe Buffay"@en;*
	dbo:occupation	*dbr:Massage;*
	dbo:occupation	*dbr:Musician;*
	dbp:family	*"Frank Buffay, Jr."@en;*
	dbp:gender	*"Female"@en;*
	dbp:nationality	*dbr:United_States.*
dbr:David_Schwimmer	*a*	*dbo:Person;*
	rdfs:label	*"David Schwimmer"@en;*
	dbo:birthDate	*"1966-11-02";*
	dbo:occupation	*"Actor"@en;*
	dbp:spouse	*dbr:Zoe_Buckman;*
	dbo:education	*dbr:Northwestern_University.*
dbr:Jennifer_Aniston	*a*	*dbo:Person;*
	rdfs:label	*"Jennifer Aniston"@en;*
	dbo:birthDate	*"1969-02-11";*
	dbo:occupation	*"Actress"@en;*
	dbp:spouse	*dbr:Brad_Pitt;*
	dbo:education	*dbr:Fiorello_H._LaGuardia_High_School.*
dbr:Courteney_Cox	*a*	*dbo:Person;*
	rdfs:label	*"Courteney Cox"@en;*
	dbo:birthDate	*"1964-06-15";*
	dbo:occupation	*"Actress"@en;*
	dbp:spouse	*dbr:David_Arquette;*
	dbo:education	*dbr:Mount_Vernon_Seminary_and_College.*
dbr:Matthew_Perry	*a*	*dbo:Person;*
	rdfs:label	*"Matthew Perry"@en;*
	dbo:birthDate	*"1969-08-19";*
	dbo:occupation	*"Actor"@en;*
dbr:Matt_LeBlanc	*a*	*dbo:Person;*
	rdfs:label	*"Matt LeBlanc"@en;*
	dbo:birthDate	*"1967-07-25";*
	dbo:occupation	*"Actor"@en;*
	dbp:spouse	*"Melissa McKnight"@en;*
dbr:Lisa_Kudrow	*a*	*dbo:Person;*
	rdfs:label	*"Lisa Kudrow"@en;*
	dbo:birthDate	*"1963-07-30";*
	dbo:occupation	*"Actress"@en;*
	dbp:spouse	*"Michel Stern"@en;*
	dbo:almaMater	*dbr:Vassar_College;*

The SPARQL query evaluation starts with the SPARQL graph pattern in the WHERE *clause. The graph pattern is a BGP of three triple patterns, can be described as* $Q = (tp_1, tp_2, tp_3)$*, where*

$tp_1 = (?character\ dbo{:}portrayer\ ?actor)$
$tp_2 = (?character\ dbo{:}occupation\ ?job)$
$tp_3 = (dbr{:}Rachel\_Green\ dbp{:}spouse\ ?character)$

According to Definition 2.10, we evaluate the query Q *over the given RDF graph denoted as* G *as follows:*

$$[\![Q]\!]_G = ([\![tp_3]\!]_G \bowtie [\![tp_1]\!]_G \bowtie [\![tp_2]\!]_G)$$

To evaluate the input query Q*, first, we evaluate each individual triple pattern over*

the graph G, generating solution mappings for each tp as follows:

$\Omega_1 = \{\omega_1 = \{character \rightarrow dbr{:}Ross\_Geller, actor \rightarrow dbr{:}David\_Schwimmer\},$
$\omega_2 = \{character \rightarrow dbr{:}Rachel\_Green, actor \rightarrow dbr{:}Jennifer\_Aniston\},$
$\omega_3 = \{character \rightarrow dbr{:}Monica\_Geller, actor \rightarrow dbr{:}Courteney\_Cox\},$
$\omega_4 = \{character \rightarrow dbr{:}Chandler\_Bing, actor \rightarrow dbr{:}Matthew\_Perry\},$
$\omega_5 = \{character \rightarrow dbr{:}Joey\_Tribbiani, actor \rightarrow dbr{:}Matt\_LeBlanc\},$
$\omega_6 = \{character \rightarrow dbr{:}Phoebe\_Buffay, actor \rightarrow dbr{:}Lisa\_Kudrow\}\}$

$\Omega_2 = \{\omega_7 = \{character \rightarrow dbr{:}Ross\_Geller, job \rightarrow dbr{:}Paleontology\},$
$\omega_8 = \{character \rightarrow dbr{:}Rachel\_Green, job \rightarrow dbr{:}Senior\_management\},$
$\omega_9 = \{character \rightarrow dbr{:}Monica\_Geller, job \rightarrow dbr{:}Chef\},$
$\omega_{10} = \{character \rightarrow dbr{:}Chandler\_Bing, job \rightarrow dbr{:}Statistical\_inference\},$
$\omega_{11} = \{character \rightarrow dbr{:}Joey\_Tribbiani, job \rightarrow "Actor"@en\},$
$\omega_{12} = \{character \rightarrow dbr{:}Phoebe\_Buffay, job \rightarrow dbr{:}Massage\},$
$\omega_{13} = \{character \rightarrow dbr{:}David\_Schwimmer, job \rightarrow "Actor"@en\},$
$\omega_{14} = \{character \rightarrow dbr{:}Jennifer\_Aniston, job \rightarrow "Actress"@en\},$
$\omega_{15} = \{character \rightarrow dbr{:}Courteney\_Cox, job \rightarrow "Actress"@en\},$
$\omega_{16} = \{character \rightarrow dbr{:}Courteney\_Cox, job \rightarrow "Actress"@en\},$
$\omega_{17} = \{character \rightarrow dbr{:}Matthew\_Perry, job \rightarrow "Actor"@en\},$
$\omega_{18} = \{character \rightarrow dbr{:}Matt\_LeBlanc, job \rightarrow "Actor"@en\},$
$\omega_{19} = \{character \rightarrow dbr{:}Lisa\_Kudrow, job \rightarrow "Actor"@en\}\}$

$\Omega_3 = \{\omega_{20} = \{character \rightarrow dbr{:}Ross\_Geller\}$

The second step in the process of computing the solution mappings of the BGP is to apply the $\bowtie$ operator on the computed sets of solutions mappings from the individual triple patterns. To execute a join operator $\bowtie$, we join the compatible mappings based on a left-linear plan as shown in Figure 2.3. For instance, we begin with joining $\Omega_1 \bowtie \Omega_3$, the solution mappings $\omega_{20} \in \Omega_3$ and $\omega_1 \in \Omega_1$ are compatible $\omega_{13} \| \omega_1$ since $dom(\omega_{20}) \cap dom(\omega_1) = actor$, and $\omega_{13}(actor) = \omega_1(actor) = dbr : Ross\_Geller$. On the other hand, the rest of the combinations are incompatible, failing to perform the join operator.

Similarly, we join the output solution mapping of ($\Omega_4 = \Omega_1 \bowtie \Omega_3 = \{\omega_{14} = \{character \rightarrow dbr : Ross\_Geller, actor \rightarrow dbr : David\_Schwimmer\}\}$) with the solution mappings of the second triple pattern Ω_2 to compute the final solution mappings, as follows: $\Omega_5 = ((\Omega_1 \bowtie \Omega_3) \bowtie \Omega_2$ resulting in:

$\Omega_5 = \{\omega_{15} = \{character \rightarrow dbr{:}Ross\_Geller, actor \rightarrow dbr{:}David\_Schwimmer,$
$job \rightarrow dbr{:}Paleontology\}\}$

The final step is to project all variables in the BGP to retrieve the answers of the SPARQL query, as follows:

$\textsc{Select}_*(\Omega_5) = \{\{character \rightarrow dbr{:}Ross\_Geller,$
$actor \rightarrow dbr{:}David\_Schwimmer, job \rightarrow dbr{:}Paleontology\}\}$

2.6 HDT

Traditional RDF serializations (e.g. RDF/XML [DMvH+00], Notation3 (N3) [BCK+08], JSON-LD [LG12], and Turtle [BB08]) have several limitations in terms of publishing and exchanging RDF graphs: traditional RDF representations (i) rely on textual representation which is quite a verbose and space-inefficient format, leading to large data transfer over the network; (ii) lack standardized metadata to give a brief summary (e.g.statistical and editorial information) to describe the represented RDF graph; (iii) demand in the worst case an expensive full scan on the entire RDF graph to perform triple lookup operations such as triple pattern matching.

In the following, we present HDT [FMG+13] which is a compact efficient representation that addresses the aforementioned limitations. It is worthwhile to mention that HDT [FMG+13] binary compression format for RDF graphs is the "under the hood" RDF format in currently existing Web querying interfaces, namely TPF [VSH+16], (br)TPF [HA16], SaGe [MSM19], SPF [AKMH20] which we will review comprehensively in Chapter 3. We also rely on HDT in our own approaches `smart-KG` [AFA+20] and `WiseKG` [AAM+21] which we present in Chapter 4 and Chapter 5, respectively.

What is HDT? HDT [FMG+13] is a compressed format for RDF graphs, which permits efficient triple pattern retrieval over the compressed data. HDT offers search and retrieval over the compressed RDF graphs without the need for decompression, and HDT provides query-relevant statistics in its metadata. It is designed to be a compact and efficient representation of RDF data that can be used for the storage, transmission, and processing of large-scale RDF datasets. HDT offers an RDF serialization format with the following advantages over the conventional RDF formats:

- HDT provides a compact format, thereby saving disk space, reducing network traffic, and downsizing the RDF graph data transfer volume.
- HDT incorporates standard metadata about the represented RDF graph in the header of the HDT file offering efficient and modular access to a detailed summary of the RDF graph, including query-relevant statistics such as number of triples, number of predicates, and number of distinct Subjects and Objects.
- HDT delivers a set of basic search operations such as fast lookups and retrieval which are essential for efficient triple pattern join operations, granting efficient querying capabilities on compressed RDF graphs.

HDT Components. HDT consists of three main components: (H)eader, (D)ictionary, and (T)riples, as follows:

- A Header component, which provides descriptive metadata (publishing information, basic statistics, etc.) about the RDF graph;

- A Dictionary component, which maps RDF terms to a compact representation of unique IDs. The main motive for this mapping is to compress the RDF triples by substituting long and repeated textual representations of the RDF terms with short and compact IDs;
- A Triples component, which encodes the resulting ID-graph (i.e. a graph of ID-triples after replacing RDF terms by their corresponding dictionary IDs) as a bitmap-encoded set of adjacency lists, one per different subject in the graph.

HDT Utilities. HDT dictionary and triples are self-indexed to support efficient retrieval operations. The dictionary implements prefix-based Front-Coding compression [MBC+16], which allows for high compression ratios and efficient string-to-id and id-to-string operations. The main compression idea relies on ordering triples by subject-predicate-object (SPO) and grouping repetitive RDF terms. An HDT file could be viewed as a compressed and directly queryable SPO-ordered bitmap-based index [FMG+13].

HDT compressed graphs are typically enriched with a companion HDT index file [MGF12] which is a compressed binary utility index built upon loading time. This additional file includes two inverted indexes on the ID-triples (in OPS and PSO order) to achieve high performance for resolving all SPARQL triple patterns. In addition, RDF graphs compressed with HDT can be queried and loaded in memory or mapped from disk without prior decompression [ÁBFM11]. HDT exhibits competitive performance for scan queries as well as triple pattern execution when the subject is provided.

The HDT library[30] offers a wide range of functionality for working with HDT data, including generating HDT files from RDF datasets, loading and saving HDT files, and performing various operations on the data contained within HDT files. HDT and HDTManager are central components of the HDT Library where they provide a convenient interface for working with HDT files and allow developers to easily incorporate HDT functionality into their applications. In this thesis, we utilize a variety of operations from the HDT library in our implementations:

- HDTManager.generateHDT(): generates an HDT file from an input RDF dataset file.
- HDTManager.mapHDT(): maps an entire HDT file to the memory to provide efficient data access. However, this method can be memory-intensive and may not be suitable in the case of large HDT files.
- HDTManager.mapIndexedHDT(): maps an HDT file into memory by only loading the index structures of the HDT file to provide fast access to specific triples based on the subject, predicate, and object elements. This method is more memory-efficient than HDTManager.mapHDT(), as it does not load the entire HDT file into memory.

[30] https://www.rdfhdt.org/development/

However, it may not be as fast as HDTManager.mapHDT() for certain types of queries since it requires accessing the disk.

- HDTManager.loadHDT(): loads an HDT file into memory. It takes the path to an HDT file as input and returns an HDT object. The HDT object is used to perform operations such as triple pattern matching.
- saveToHDT(): saves an HDT object to a specified file on the hard disk.
- HDT.search(): performs triple pattern matching on an HDT file and returns a stream of triples that match the input pattern.
- HDT.iterate(): iterates through all the triples in an HDT file and returns them one by one.
- HDT.size(): returns the number of triples in an HDT file.

In the following, we detail an example of each component of Friends RDF graph from Example 7 in HDT format:

Example 8. *We first demonstrate the header component of Friends graph, as follows:*
<http://purl.org/HDT/hdt#HDTv1>
ntriples length=1795
<file:///friends.ttl> <http://purl.org/HDT/hdt#Dataset> .
<file:///friends.ttl> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://rdfs.org/ns/void#Dataset> .
<file:///friends.ttl> <http://rdfs.org/ns/void#triples> "79" .
<file:///friends.ttl> <http://rdfs.org/ns/void#properties> "17" .
<file:///friends.ttl> <http://rdfs.org/ns/void#distinctSubjects> "12" .
<file:///friends.ttl> <http://rdfs.org/ns/void#distinctObjects> "60" .
<file:///friends.ttl> <http://purl.org/HDT/hdt#statisticalInformation>_:statistics.
<file:///friends.ttl> <http://purl.org/HDT/hdt#publicationInformation> _:publicationInformation.
<file:///friends.ttl> <http://purl.org/HDT/hdt#formatInformation> _:format .

_:format <http://purl.org/HDT/hdt#dictionary> _:dictionary .
_:format <http://purl.org/HDT/hdt#triples> _:triples .

_:dictionary <http://purl.org/dc/terms/format>
<http://purl.org/HDT/hdt#dictionaryFour> .
<http://purl.org/HDT/hdt#dictionarynumSharedSubjectObject> "8" .
<http://purl.org/HDT/hdt#dictionarymapping> "1" .
<http://purl.org/HDT/hdt#dictionarysizeStrings> "2047" .
<http://purl.org/HDT/hdt#dictionaryblockSize> "16" .

```
_:triples <http://purl.org/dc/terms/format>
<http://purl.org/HDT/hdt#triplesBitmap> .
_:triples <http://purl.org/HDT/hdt#triplesnumTriples> "79" .
_:triples <http://purl.org/HDT/hdt#triplesOrder> "SPO" .

_:statistics <http://purl.org/HDT/hdt#originalSize> "4041" .
_:statistics <http://purl.org/HDT/hdt#hdtSize> "2438" .

_:publicationInformation <http://purl.org/dc/terms/issued> "2022-09-15T13:07:18".
```

As shown in Example 8, the header component is described in turtle format. HDT extends VoiD [ACHZ09], the standard vocabulary for describing metadata about RDF datasets. In particular, HDT uses the namespace `http://purl.org/HDT/hdt#hdt` to describe the properties of RDF HDT datasets. In the following, we describe the header structure of the Friends graph:

- **VoiD**: the header use Void properties [ACHZ09] to add provenance to the dataset in a standard way. For instance, HDT captures basic statistical metadata about the dataset using the following properties #triples, #properties, #distinctSubjects, #distinctObjects.
- **#publicationInformation**: groups information about the HDT file. Our example shows the issue date of the dataset.
- **#statistical metadata**: includes basic statistical information to give an overview of the materialized dataset. In our example, it states the original graph size, #originalSize ≈ 4KB, and the hdt graph size, #originalSize ≈ 2.5KB.
- **#formatInformation**: groups concrete information about the dictionary and the triples component. This metadata is utilized during the retrieval operations. In our example, hdt:dictionary contains four elements to describe the concrete implementation of the dictionary, for example, #dictionarynumSharedSubjectObject which describes the number of entities in the graph that appear in both subject and object positions. In addition, hdt:triples contains three elements to characterize the triples representation. For our Friends graph 7, #triplesBitmap describes that the triples are physically materialized in a bitmap format.

To conclude, several Web querying interfaces rely on HDT as a backend data structure as we will explain in Chapter 3. TPF [VSH+16] and SPF [AKMH20] rely on an HDT of the whole graph G to evaluate triple patterns on the server with a low computation footprint,

whereas smart-KG [AFA+20] and WiseKG [AAM+21] profit from the compression and also lowering the network footprint.

CHAPTER 3

A uniform characterization of existing Web querying interfaces

The client-server query processing model [FJK96] typically distributes query workload between the service provider (i.e a powerful centralized server) and service requester (i.e. clients). In the context of SPARQL Web querying, workload distribution among clients and servers is typically constituted by query processing shipping strategies which dictate the processing location of the query graph patterns. Neither query nor data shipping strategies enable reliable, efficient, and low-cost query execution [VSH+16]. To address this, the Linked Data fragments (LDF) framework has been proposed to provide a uniform view of potential SPARQL Web interfaces which distribute query evaluation load among clients and servers using hybrid shipping strategies. In this chapter, we explore the existing hybrid shipping-based interfaces following the LDF framework characterization, with the aim of offering live SPARQL querying evaluation over large-scale knowledge graphs while efficiently sharing the computation cost among clients and servers.

3.1 Linked Data Fragments framework

In the following, we define query interfaces for KGs following the principles set by the Linked Data Fragments framework (LDF) [VSH+16]. We borrow from the original specification [VSH+16] and align formal definitions and notations to uniformly present different APIs. The introduced formalization enables us to establish the possibility of returning fragments either as a single graph or as one subgraph for each solution, a scenario not permissible under the existing LDF characterization. Additionally, the LDF characterization places emphasis on metadata and hypermedia controls, whereas our approach disregards these specifics and instead concentrates on devising a comprehensive, uniform definition for diverse partitioning techniques to be employed as fragments.

Definition 3.1 (LDF API, adapted from [VSH+16]). *An LDF API of a KG G accessible at an endpoint URI u* [a] *is a tuple $f = \langle s, \Phi \rangle$ with*

- *a selector function $s(G, Q, \Omega)$ that given a query pattern Q and (a potential entry) set of bindings Ω, it returns a fragment $\Gamma \subseteq G$, or alternatively a set of fragments*[b] *$\Gamma^* \subseteq 2^G$,*
- *a paging mechanism $\Phi(n, l, o)$ parameterized by $n, l, o \in \mathbb{N}_0$ denoting maximum page size, limit, and offset.*

[a] Via this base URI the API can be accessed and queried as well as additional controls can be submitted.

[b] We note that this is a generalization from the original LDF proposal, which – technically – could be realized, for instance, by returning RDF datasets in the sense of SPARQL (consisting of a default graph and optionally a set of (named) graphs), or resp. a set of quads instead of triples.

In essence, LDF characterizes APIs that allow access to fragments of a KG G through (specific to a particular instantiation of LDF) a limited range of allowed query patterns that a client can submit to the server; often with the goal to limit server-side computation cost and to enable effective HTTP caching while leaving evaluations of more complex patterns to the client. Variations of LDF also offer additional controls to ship intermediate bindings alongside queries or to control the "chunk size" of results by specifying page sizes into which the results should be batched.

The selector function s has as parameters an RDF graph G, a SPARQL pattern Q, and a set of bindings Ω,[1] where we define two variants, $s(\cdot)$ and $s^*(\cdot)$, which differ essentially in terms of returning either a single graph containing the union of all triples relevant to *any* solution or one subgraph *per* solution $\omega \in [\![Q]\!]_G$:

Definition 3.2 (Function s). $s(G, Q, \Omega) = \{t \in \omega(Q) \mid \omega \in [\![Q]\!]_G : G \models \omega(Q) \wedge (\exists \omega' \in \Omega : \omega' || \omega)\}$

Definition 3.3 (Function s^*). $s^*(G, Q, \Omega) = \{\omega(Q) \mid \omega \in [\![Q]\!]_G : G \models \omega(Q) \wedge (\exists \omega' \in \Omega : \omega' || \omega)\}$

All LDF APIs, we present in the following, can be expressed in terms of one of these two default selector functions. Note that, whenever the set of bindings Ω is not considered (i.e. only empty binding sets $\Omega = \emptyset$ are expected) in a particular selector function, we will conveniently also just write short $\sigma(G, Q)$ (or $s(G, Q)$, $s^*(G, Q)$, resp.) instead of $\sigma(G, Q, \Omega)$ (or $s(G, Q, \Omega)$, $s^*(G, Q, \Omega)$, resp.) in the following. As we will see, all existing

[1] We note that this strict definition of allowed parameters for s is not made in [VSH+16], but we will rather use those here to describe the considered APIs uniformly.

LDF APIs considered in this paper and summarised in Table 3.1 can be expressed in terms of one of the two standard selector functions $s(\cdot)$ and $s^*(\cdot)$, whereas we will extend and modify those – in terms of partition-based LDF – in the following.

The general paging mechanism Φ we use in this thesis enables the ability to retrieve the result in batches e.g., for the cases where Γ (or, resp., Γ^*) is overly large; only partial results are required or to enable incremental result. Hence, we assume that $\Phi(n, l, o)$ simply defines a mechanism to divide Γ into a set of partitions (or *pages*) $\Gamma^* = \{\Gamma_0, \ldots, \Gamma_{k-1}\}$, where for each page Γ_i it is guaranteed that $|\Gamma_i| < n$ (i.e., Γ_i does not contain more than n triples), and l and o, resp. would allow requesting the pages from Γ_o to Γ_{o+l-1}[2]. We assume l to default to $l = 1$, o to default to $o = 0$, and finally $n = \infty$ signifying that whole graph G or $\Gamma = G$ (or, $\Gamma^* = \{G\}$) should be returned. As such l, o should be viewed analogous to the SPARQL LIMIT and OFFSET modifiers but applied to pages instead of individual solution mapping.

Herein, we provide a characterization of SPARQL Web querying interfaces from the literature within the spectrum of the LDF framework. We summarize their respective characterizations in Table 3.1 and describe the existing implementations of LDF in the following subsections.

3.1.1 Data Dump

A Data Dump may be considered a full client-side LDF interface where the data publisher offers RDF data dump in a downloadable form and the full query processing is done offline on the client-side.

Definition 3.4. *(Data Dumps)*

Data dumps can be characterized in terms of LDF, as follows:

- *the selector function $s(\cdot)$ as defined in Definition 3.2,*
- *the only admissible form of Q and Ω are $Q = \{(?s, ?p, ?o)\}$ and $\Omega = \emptyset$, i.e., $s(G, Q, \Omega)$ boils down to the identity function,*
- *Φ: the only admissible parameter for $\Phi(n, l, o)$ is $\Phi(\infty, 1, 1) = \{\Gamma_1\} = \{\Gamma\}$.*

To perform a SPARQL query, clients request an entire KG from the server and deploy a SPARQL engine to locally process their queries. A use case where data dumps can be a very valuable solution is when clients have powerful processing resources while demanding resource-hungry query workload tasks. However, in general, the data dump solution puts the processing cost on the clients, plus incurs potentially high network traffic on both client and server sides in the case of frequently evolving KGs.

[2] As such l,o should be viewed synonymously with SPARQL's LIMIT and OFFSET modifiers.

Table 3.1: Aligned formal definitions and notations with LDF original specifications to uniformly present different existing LDF APIs

LDF Interface	Definition
Data Dump	The selector function is $s(\cdot)$ The only admissible form of Q and Ω are $Q = \{(?s, ?p, ?o)\}$ and $\Omega = \emptyset$ The only admissible parameter for $\Phi(n, l, o)$ is $\Phi(\infty, 1, 0) = \{\Gamma_1\} = \{\Gamma\}$
TPF	The selector function is $s(\cdot)$ The only admissible form of Q are triple patterns and $\Omega = \emptyset$ $\Phi(n, l, o)$ allows results to be "batched" into chunks of n triples, i.e., in TPF the publisher can set n as a parameter, whereas limit $l = 1$ as it is possible to only retrieve one page at a time, and offset o is the page number requested by the client
brTPF	The selector function is $s(\cdot)$ The only admissible form of Q are triple patterns Ω can be arbitrary $\Phi(n, l, o)$ as defined in TPF
SPF	The selection $s^*(G, Q, \Omega)$, i.e., $s^*(\cdot)$ is used to return results per pattern solution the only admissible form of Q are star-shaped BGPs Ω can be any set of bindings $\Phi(n, l, o)$: as solutions are returned per pattern solution, n is fixed to the star pattern of size k but SPF also allows to paginate over solutions, i.e., retrieving results in chunks of l (iterating over increasing offsets $o := o + l$)
SPARQL Endpoints	A variant of $s^*(\cdot)$ by returning subgraphs of the form $\omega(Q)$. In practice, SPARQL endpoints return solution mappings, yet, it is possible to devise a correspondence between these and $s^*(\cdot)$ Any pattern Q is admissible $\Omega = \emptyset$, unless `VALUES` patterns are considered. In this case, Ω is encoded in the `VALUES` clause of Q Φ: the standard `LIMIT` and `OFFSET` operators for BPGs could be considered as `LIMIT` l and `OFFSET` o such that $n = 1$
SAGE	A variant of $s^*(\cdot)$ by returning subgraphs of the form $\omega(Q)$, analogous to SPARQL endpoints. Any pattern Q is admissible $\Omega = \emptyset$, unless `VALUES` patterns are considered. In this case, Ω is encoded in the `VALUES` clause of Q $\Phi(1, \infty, o)$: Assuming that Q does not include the keywords `LIMIT` and `OFFSET`. o is used to indicate that $\{\Gamma_1, \ldots, \Gamma_{o-1}\}$ has been received by the client. In practice, the SAGE client sends the last solution mapping ω that has been produced in Γ_{o-1}. Still, it is possible to devise a correspondence between ω and o. $\Phi(1, l, o + o')$: Assuming that Q does include the keywords `LIMIT` l and `OFFSET` o'. o is defined as in the previous case

3.1.2 SPARQL endpoint

A SPARQL endpoint is a server-side solution that minimizes the query processing load on the client-side which only receives the final results of the submitted query. RDF KGs are traditionally exposed via SPARQL endpoints, i.e., APIs that serve full SPARQL queries over the HTTP protocol [FWCT13]. We can also understand any SPARQL endpoint as an LDF interface in our introduced terminology.

Definition 3.5. *(SPARQL endpoint)*

SPARQL endpoints can be characterized in terms of LDF as follows:

- *while SPARQL endpoints usually directly return sets of bindings, they can also be viewed as a variant of $s^*(\cdot)$ by returning subgraphs of the form $\omega(Q)$[a],*
- *any pattern Q is admissible;*
- *$\Omega = \emptyset$, unless `VALUES` patterns are considered, which could be viewed as equivalent to binding restrictions a la LDF,*
- *Φ: while some SPARQL endpoints support other forms of paging, the standard LIMIT and OFFSET operators for BPGs could be considered as `LIMIT` l and `OFFSET` o such that $n = |Q|$; however, note that subsequent calls of SPARQL queries with consecutive OFFSETs are in general not guaranteed to behave deterministically.*

[a]Deriving ω is straightforward since, given Q, ω and $\omega(Q)$ are in a trivial 1-to-1 correspondence. We prefer this interpretation of the LDF metaphor to SPARQL endpoints over – as suggested in a side note in [VSH+16] – relying on encoding result sets as RDF triples (such as using e.g. the informal RDF SPARQL result format from the SPARQL1.1 Test Case Structure, cf. `https://www.w3.org/2009/sparql/docs/tests/README.html`) since the latter would not return subgraph(s) of G.

Public SPARQL endpoints for various KGs such as DBpedia, YaGo [SKW07], and Bio2RDF [DCC+14] often run on top of RDF triples stores such as Virtuoso [EM09] and Blazegraph [3]. Virtuoso runs DBpedia, YaGo, and Bio2RDF endpoints, while Blazegraph runs the Wikidata [VK14] endpoint. These SPARQL endpoints provide high performance under low loads. However, with concurrent clients and query complexity, endpoints face overloads and large delays that lead to well-known problems of low availability [AHUV13] and poor performance [VSH+16]. Thus, most SPARQL endpoints turn into resource-hungry services, too costly to host and maintain for data providers. Latest studies on public SPARQL endpoints [AHUV13, PKF+20] confirm these issues and show that at least half of the endpoints do not answer at all, while others impose significant restrictions such as refusing complex queries or limiting result sizes [APU14]. In practice, public SPARQL endpoints often impose limitations on the execution of SPARQL queries in order to to ensure a balanced distribution of server resources among clients. These limitations may include restrictions on the execution time of queries, the number of results returned per query, and the rate of queries per IP address. For instance, the DBpedia SPARQL endpoint [4] allows queries to run for a maximum of 120 seconds and returns no more than 10000 results. It also limits the number of parallel connections to 50 and the number of

[3]`https://blazegraph.com/`
[4]`http://wiki.dbpedia.org/public-sparql-endpoint`

HTTP requests per second per IP address to 100. The Wikidata SPARQL endpoint [5] has even stricter limitations, setting the maximum execution time for SPARQL queries at 60 seconds.

3.1.2.1 SaGe

SaGe [MSM19] is a SPARQL query engine tailored to address the undesirable starvation of simple queries waiting for long-running queries to release the server resources. To this aim, SaGe proposes a preemptive Web server. SaGe may be viewed as an extension of SPARQL endpoints which introduces a round-robin scheduling mechanism that allocates a fixed time quantum per query: once a query is executed for that given quota, the query is suspended and resumed upon client request. Then, SaGe server proceeds with the next waiting query. To resume queries, SaGe ships to the client the state of the query execution.

SaGe [MSM19] may be considered a variant of general SPARQL endpoints that supports Web preemption in order to guarantee a more fair distribution of server resources amongst concurrent clients. Under Web preemption, the server suspends a running query Q after a predefined time quantum τ and returns partial results $\{\Gamma_1, \ldots, \Gamma_{o-1}\}$ to the client. A SPARQL query Q is resumed based on the client's request; this process is repeated until all results are produced. This ability enables SaGe to prevent long-running queries from exploiting the server resources, especially under high concurrent load [MSM19]. The client can then (with additional hypermedia controls) deterministically continue exactly at offset o in a subsequent call.

Definition 3.6. *(SaGe)*

SaGe can be characterized in terms of LDF, as follows:

- *a variant of $s^*(\cdot)$ by returning subgraphs of the form $\omega(Q)$.*
- *any pattern Q is admissible.*
- *$\Omega = \emptyset$, unless `VALUES` patterns are considered. In this case, Ω is encoded in the `VALUES` clause of Q.*
- *$\Phi(1, \infty, o)$: Assuming that Q does not include the keywords `LIMIT` and `OFFSET`. o is used to indicate that $\{\Gamma_1, \ldots, \Gamma_{o-1}\}$ has been received by the clientİn practice, the SAGE client sends the last solution mapping ω that has been produced in Γ_{o-1}. Still, it is possible to devise a correspondence between ω and o.*

[5] https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual#Query_limits

- $\Phi(1, l, o+o')$: *Assuming that Q does include the keywords* `LIMIT` l *and* `OFFSET` o'. o *is defined as in the previous case.*

The experimental evaluation [MSM19] demonstrates that SaGe improves the average time required to receive the first result and the average workload completion time per client compared to traditional SPARQL endpoints under concurrent query load. In general, SaGe has impressive performance for most of the query shapes. However, SaGe suffers from excessive delays and a high number of requests in the case of high concurrent clients with complex queries due to frequent query context switching.

3.1.3 Triple Pattern Fragment (TPF)

TPF [VSH+16] is an interface to enable reliable querying over KGs by limiting the server functionality to only answer single triple patterns and delegating the processing of the resource-demanding more complex patterns – and particularly joins – to the client-side.

Definition 3.7. *(TPF)*

In terms of the generic LDF framework, TPF is the most straightforward "incarnation", defined as in

- *the selector function is $s(\cdot)$ as defined above,*
- *the only admissible form of P are triple patterns and $\Omega = \{\emptyset\}$,*
- $\Phi(n, l, o)$*: allows results to be "batched" into chunks of n triples, whereas limit l and offset o cannot be set explicitly in TPF.*

TPF clients receive paginated intermediate results of each triple pattern in the query and incrementally combine the intermediate results to compute the complete results on the client. While experimental results [HAMS18, VSH+16] show that TPF achieves higher server availability than traditional server-centric SPARQL endpoints, this typically comes at the expense of a significant increase in the network traffic due to considerable overheads from HTTP requests and data transfers. In particular, non-selective queries ((i.e. queries with high cardinality triple patterns) can be penalized by a high number of irrelevant intermediate results transferred (i.e. transferred data that does not contribute to the final query answer) and costly client-side join operations.

3.1.4 Binding-Restricted Triple Pattern Fragment

Bindings-Restricted Triple Pattern Fragments [HA16] (brTPF) gives a slight boost to the performance of TPF by attaching intermediate results to triple pattern requests along with distributing the join between the client and the server using the bind join strategy [HKWY97]. In essence, brTPF [HA16] is an extension of TPF that additionally

permits arbitrary $\Omega \neq \emptyset$. The attached solution mappings Ω from the previously evaluated triple patterns potentially reduce the number of HTTP requests and data received with respect to the original TPF solution [HA16]. However, the number of requests is still relatively high, combined with the need to transfer these intermediate results verbatim.

Experiments [HA16] show that brTPF provides an overall better query performance than TPF. As we will see, though, brTPF still potentially encounters serious delays with increasing the number of concurrent clients or with queries that require shipping a large number of intermediate results.

Definition 3.8. *(brTPF)*

brTPF can be characterized in terms of LDF, as follows:

- *The selector function is* $s(\cdot)$
- *The only admissible form of* Q *are triple patterns*
- Ω *can be arbitrary*
- $\Phi(n, l, o)$ *as defined in TPF*

3.1.5 Star Pattern Fragments (SPF)

SPF [AKMH20] proposes to generalize brTPF from single triples to handling star-shaped subqueries on the *server*. SPF splits BGPs of a given query into star-shaped sub-queries where each subquery consists of triple patterns that share the same subject. Similar to TPF, more complex queries involving joins over stars or single triples are processed on the client. Still, evaluating star-shaped subqueries directly on the server may drastically reduce the number of requests made during query processing while still maintaining a relatively low server load since star patterns can be answered relatively efficiently by the server [PAG09]. For processing joins efficiently, analogously to brTPF, bindings can be shipped along with each star-shaped subquery. SPF, as an instance of LDF, differs from brTPF with respect to the restriction of the selector function and allowed patterns:

Definition 3.9. *(SPF)*

SPF can be characterized in terms of LDF as follows:

- $s_{SPF}(G, P, \Omega) = s^*(G, P, \Omega)$*, i.e.,* $s^*(\cdot)$ *is used to return results per pattern solution,*
- *the only admissible form of* P *are star-shaped BGPs,*

- Ω *can be any set of bindings,*
- $\Phi(n, l, o)$*: as solutions are returned per pattern solution,* n *is fixed to the star pattern of size* k *but SPF also allows paginating over solutions, i.e., retrieving results in chunks of* l *(iterating over increasing offsets* $o := o + l$*).*

Experiments [AKMH20] show that SPF (compared to brTPF) decreases the average network load. However, as we will see, SPF suffers from relatively high server CPU usage.

3.2 Partition-based LDF

Unlike the previous LDF approaches, which – given a particular admissible query Q – would return a graph or partition that exactly contains the query results, in this paper, we will focus on an alternative approach that rather will ship an overestimate from a set of hosted partitions that potentially can be used to answer the query, which we will call *partition-based LDF* approaches: partition-based LDF can be seen as a generalization of the aforementioned, existing LDF interfaces, which - instead of the exact triples matching a particular admissible query pattern Q, rather returns a subset of partitions from a pre-computed partitioning $\mathcal{G} = \{G_1, \ldots, G_n\}$ of G, such that $G = G_1 \cup G_2 \cup \cdots \cup G_m$, where $\forall i \neq j \ G_i \cap G_j = \emptyset$. I.e,. $\mathcal{G}$ is a cover of G.

That is, the idea here is that a *partition-based LDF server* serves $\mathcal{G}$ such that upon an LDF API call with a query pattern Q the selector function $\sigma(G, Q) \subseteq \mathcal{G}$ returns a subset of matching partitions that contain all query answers for Q. That is, it is ensured that

$$[\![Q]\!]_G \equiv [\![Q]\!]_{\bigcup_{G_i \in \sigma(Q,G)} G_i} \tag{3.1}$$

hence, the client can therefore compute the complete result of the actual query Q from just calling and retrieving $\sigma(G, Q)$ from the server.

As we will see, different partitioning techniques lend themselves to this overall idea better or worse: the tricky part is to find a partitioning $\mathcal{G}$ such that (1) $\sigma(G, Q)$ can be easily computed from Q, and (2) $\sigma(G, Q)$ provides a "close estimate" minimizing the number of partitions to be shipped and where the union of these partitions does contain all necessary, but not too many unnecessary triples for computing the actual query Q, and finally (3) all possible partitions for any admissible queries Q in the range of $\sigma(Q, G)$ can be efficiently served (and, ideally, pre-computed) on and LDF-server.

In this context, we note that shipping a full **Data Dump** could be considered as a "trivial" partitioning-shipping technique, where

- $\mathcal{G} = \{G\}$
- any query Q is admissible

- $\sigma(G, Q) = \{s(G, \{?s, ?p, ?o\})\} = \mathcal{G}$

As such, non-trivial partition-based LDF methods could be considered as shipping only a "necessary subset of partial dumps per query". Further, for partition-based LDF interfaces, in general, we herein will assume

- $\Omega = \emptyset$ is the only admissible binding set, i.e., we do not consider binding restrictions,
- Φ: only $n = \infty$ is admissible, i.e., no paging is supported since the union of all relevant partitions will be typically needed to compute the query results.

Our proposed partition-based LDF approach will be extensively elaborated and evaluated in the forthcoming three chapters.

CHAPTER 4

Hybrid Shipping for SPARQL Querying on the Web

In the preceding chapter, we presented an overview of Linked Data framework and several approaches that adopt this framework, including Triple Pattern Fragments (TPF) and brTPF. While these approaches enhanced the server availability compared to SPARQL endpoints, they have limitations that make it challenging to maintain a live public SPARQL querying service for web-scale KG as they may suffer from a significant decline in the performance of SPARQL queries and unnecessary transfer of irrelevant data on complex queries with large intermediate results.

In this chapter, we introduce `smart-KG`, a hybrid shipping interface designed to enhance the server availability of Web querying interfaces while maintaining a high performance by distributing query workloads between servers and clients. Rather than solely relying on shipping intermediate results, as previous solutions such as TPF do, `smart-KG` combines TPF with the shipping of modular, query-relevant KG partitions that can be directly queried by clients locally. To achieve this, `smart-KG` server maintains compressed and queryable graph partitions, that is, KG "slices" can be shipped, cached, and locally queried by `smart-KG` clients. We propose a graph partitioning technique based on Characteristic Sets [GN14, NM11], which leverages the structure of RDF graphs to group entities described with the same sets of predicates. In addition, smart clients implement a query executor that combines KG partitions and intermediate results of triple patterns received from the server to evaluate SPARQL queries. Our evaluations demonstrate that `smart-KG` outperforms state-of-the-art solutions in terms of performance and significantly reduces data transfer volume while improving server-side availability compared to the state-of-the-art interfaces.

Chapter Organization. The remainder of this chapter is organized as follows:

- In Section 4.1, we introduce KG partition technique named family-partitioning which is designed for graphs with skewed predicate distributions to trade-off the number of partitions to be maintained and transferred. We then use HDT [FMG+13] to represent and distribute such partitions in a compressed and queryable way.
- In Section 4.2, we present a novel paradigm, `smart-KG`, to distribute the evaluation of SPARQL queries among clients and servers by leveraging the transfer of compressed KG partitions in Section.
- In Section 4.5, we provide an empirical evaluation of `smart-KG` on synthetic and real-world KGs and queries, significantly outperforming the state-of-the-art server- and client-side SPARQL query processing.
- In Section 4.6, we summarize and outline the approach limitations.

4.1 Family-Based Partitioning of RDF Graphs

RDF is a semi-structured data model which typically does not prescribe a fixed schema. In theory, this can lead to RDF graphs, where the set of predicates used to describe subjects and their relationships may vary greatly. However, in real-world RDF graphs, there typically is an inherent structure as there exist repetitions whenever subjects of the same kind are described in the same way. For instance, predicates describing *Films* (e.g., director, starring, launchDate, language, etc.) are different than those describing *Persons* (e.g., birthday, nationality, etc.) in DBpedia. In the literature, so-called characteristic sets [NM11, GN14] have been defined to capture these latent structures that eventually construct a "soft schema" from the entities that are semantically similar in a graph.

Neumann and Moerkotte [NM11, GN14] capture these structures with the notion of *characteristic sets*, also called *predicate families* [FMPdlFRG18] (or just families hereinafter). In this paper, we propose a concrete method that employs the concept of predicate families for partition-based Linked Data Fragment (LDF) interfaces, named *Family-Based Partitioning*. Let G be an RDF graph, and $subj(G), pred(G), obj(G)$ be the sets of subjects, predicates, and objects in G respectively. We define the *(predicate) family*, of a subject s, $F(s)$ as the set of predicates related to the subject s, that is:

Definition 4.1. *(Predicate Family of a Subject Term s) [NM11]*

$$F(s) = \{p \mid \exists o \in obj(G) : (s, p, o) \in G\}$$

Analogously, we denote as $F(G)$ or just F, to the set of all different *predicate* families occurring in G, as follows:

Definition 4.2. *(A set of Predicate Families) [NM11]*

$$F(G) = \{F(x) \mid x \in subj(G)\}$$

For simplicity, we name the different families in G as $F_1, F_2, ..., F_m$, where $m = |F(G)|$. In this thesis, we use families as a means to define a graph partitioning technique, i.e., we consider – as the basis for our approach – a disjoint set of partitions that is a cover[1] $\mathcal{G} = \{G_1, G_2, \cdots, G_m\}$ of G based on its families, where predicate families imply a partitioning

$$\mathcal{G} = \{G_{F_i} \mid F_i \in F(G)\} \tag{4.1}$$

usable for partition-based LDF as defined above, where each partition G_{F_i} is defined by a corresponding respective predicate familiy $F_i \in F(G)$ as follows:

Definition 4.3. *(Family Partition)*

$$G_{F_i} = \{(s, p, o) \in G \mid F(s) = F_i\}$$

We will refer to this partitioning as *family-partitioning*; slightly abusing notation we will simply write G_i for G_{F_i} in the following. Next, the admissible queries for family-partitioning are star-shaped query patterns, i.e., BGPs composed of k triple patterns form $Q = \{(s, p_i, o_i) \mid 1 \leq i \leq k,\ s \in V \cup U, p_i \in U, o_i \in V \cup U \cup L\}$ with a single common subject s, where

$$\sigma(G, Q) = \{G_i \in \mathcal{G} \mid pred(Q) \subseteq F_i\} \tag{4.2}$$

Obviously, for any star-shaped query, $\sigma(G, Q)$ contains all relevant triples from G to compute the answers.

To illustrate the previous definitions consider the KG G shown in Fig. 4.1, and the predicate families shown in Fig. 4.2. Following the definition of predicate family in Eq. (4.1), the subjects *s1* and *s2* belong to the same family F_1, as they have the same predicates. The subject *s2* belongs to family F_2. For the KG G, there are two families denoted $F(G)$, i.e., F_1 and F_2. Lastly, each of these families induces a partition over G. For example, G_{F_2} contains all the triples of subjects that belong to family F_2, which in this case is triples *t8* and *t9*. Lastly, the set of partitions computed for G, denoted $\mathcal{G}$ are G_{F_1} and G_{F_2}.

Families provide structure-based means of partitioning an RDF graph used for the following:

[1] i.e, $G = G_1 \cup G_2 \cup \cdots \cup G_m$, where $\forall i, j\ G_i \cap G_j = \emptyset$

```
:s1 rdf:type :Film . #t1
:s1 rdf:type :Work . #t2
:s1 :starring :o1 . #t3
:s1 :director :o2 . #t4
:s2 rdf:type :Work . #t5
:s2 :starring :o1 . #t6
:s2 :director :o3 . #t7
:s3 rdf:type :Work . #t8
:s3 :director :o4 . #t9
```

Figure 4.1: KG example

Predicate Families

$$
\begin{aligned}
F(:s1) &= \{rdf{:}type, director, starring\} \\
&= F_1 \\
F(:s2) &= \{rdf{:}type, director, starring\} \\
&= F_1 \\
F(:s3) &= \{rdf{:}type, director\} \\
&= F_2
\end{aligned}
$$

Predicate Families in G

$$F(G) = \{F_1, F_2\}$$

Partitions induced by each family

$$G_{F_1} = \{t1, t2, t3, t4, t5, t6, t7\}$$

$$G_{F_2} = \{t8, t9\}$$

Partitioning $\mathcal{G} = \{G_{F_1}, G_{F_2}\}$

Figure 4.2: Predicate families for the KG shown in Fig. 4.1

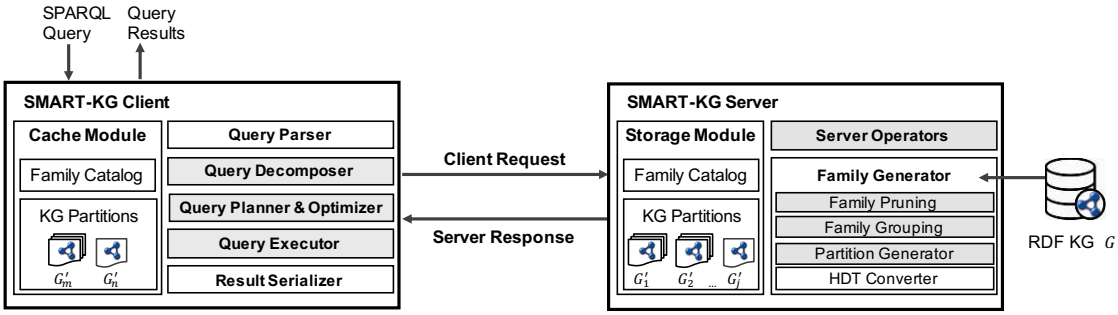


Figure 4.3: Overall architecture for the `smart-KG` client and server.

- Join and cardinality estimation [GN14, NM11] for SPARQL optimization,
- RDF compression [HMF15],
- Building indexes[2] to speed up SPARQL queries [MPMA17].

To the best of our knowledge, our work presented in [AFA+20] is the first to utilize families as a shipping strategy.

4.2 SMART-KG: Design and Overview

`smart-KG` (cf. Fig. 4.3) defines client and server operations to combine the shipping of partitions based on RDF families with the shipping of the results of evaluating triple patterns to reduce query runtime.

The `smart-KG` servers generate families and corresponding partitions of a given knowledge graph (KG). The resulting KG partitions are materialized (in HDT) in the storage module along with a family catalog that contains metadata about the structure of the partitions. In addition, `smart-KG` servers also support operators to execute triple pattern queries and transfer partitions to `smart-KG` clients.

The `smart-KG` clients are able to execute SPARQL queries by devising query plans that combine the shipping of triple pattern results and partitions. The query decomposition, planning, and optimization techniques implemented by the `smart-KG` client exploit the structure of KG partitions to reduce query execution time. In addition, `smart-KG` client partitions can be discovered, retrieved, cached, and (locally) queried by `smart-KG` clients.

4.2.1 SMART-KG Server

The `smart-KG` server, upon loading an RDF graph, supports access to graph partitions and the evaluation of triple patterns using TPF. To this end, the server implements a

[2]Meimaris et al. [MPMA17] extended the notion of characteristic sets also to object nodes

partition generator taking into consideration the families from the graph plus retrieval operations.

4.2.1.1 Partition Generator

The `smart-KG` server, upon loading a graph G, decomposes it into partitions $G_1, \dots G_m$ per family, as described in Eq. (4.3) and converts those partitions to HDT. In practice, however, the number of partitions can be relatively large for real-world RDF graphs. Thus, we introduce the concept of *predicate-restricted families*, where some particular predicates are not considered for the creation of families.

Definition 4.4. *(Predicate-restricted Family)*

Let us consider a restricted set of predicates, $P'_G \subseteq P_G$. The predicate-restricted family of a subject s w.r.t. P'_G, denoted $F'(s)$, is defined as follows:

$$F'(s) = \{p' \in P'_G \mid \exists o \in obj(G) : (s, p', o) \in G\}$$

Analogously, we denote as $F'(G) = \{F'_1, F'_2, \cdots, F'_m\}$, or just F', to the set of different predicate-restricted families for G w.r.t. P_G, where $m' = |F'(G)|$.

These families correspond to a set $\mathcal{G}' = \{G'_1, G'_2, \dots, G'_{m'}\}$ of partitions of a subgraph of G based on the P'_G-restricted families, with

Definition 4.5. *(Predicate-restricted Partition)*

$$G'_i = \{(s, p, o) \in G \mid F'(s) = F'_i\}$$

Note that, however $\mathcal{G}'$ is no longer a complete cover of G, but the graph $G' = \bigcup G'_i$ only contains the "projection" of G to P'_G.

Serving predicate-restricted families allows a `smart-KG` publisher to select P'_G depending on the following:

- The cardinality of the predicates (i.e. the number of occurrences in the graph).
- The importance of predicates (and combinations) in actual query workloads.

We will describe a concrete method to pick P'_G based on the cardinality of predicates in Section 4.2.1.3.

4.2.1.2 Family Grouping

Although the use of restricted families can control the number of generated families and avoid generating rarely used families to some extent, the number and volume of partitions are still determined further by other distribution features of the data. In practice, many RDF graphs are skewed in the sense that there exist "dominant" families with large corresponding partitions, as opposed to several small, very similar families of much smaller sizes. This phenomenon arises due to the semi-structured nature of RDF, where predicates may vary across entities of the same type.

Thus, besides using predicate-restricted families, as a second measure, our partition shipping strategy further uses merging (i.e. grouping) similar families into a single family. For instance, all disjoint families contain a certain set of predicates. An example is as follows:

F_1={`foaf:name, dbp:birthdate, dbp:title`} and F_2={`foaf:name, dbp:birthdate, dbp:occupation`} can be merged into a single family which is $F_{\{1,2\}}$={`foaf:name, dbp:birthdate`}.

The intuition behind merging such families covering overlapping predicates is that these overlapping predicate subsets may also occur as predicate families in query patterns more commonly. Therefore, instead of shipping the union of partitions contributing to a query, only the partition corresponding to the smallest merged families needs to be shipped.

Note that, in order to define the notion of a merge of families and respective (predicate-restricted) partitions we refer to particular families in $F'(G) = \{F'_1, F'_2, \cdots, F'_{m'}\}$, by their index $\{1, \ldots, m'\}$. Using this notation, formally, for an index set $I \in 2^{\{1,\cdots,m'\}}$, we define the merge F'_I of the set of families $\{F'_j \mid j \in I\}$ as follows[3]:

$$F'_I = \bigcap\nolimits_{i \in I} F'_i \tag{4.3}$$

Analogously, the corresponding merged partition $G'_I \subseteq \bigcup_{i \in I} G'_i$ can also be defined as:

$$G'_I = \{(s, p, o) \in G \mid F(I) \subseteq F(s)\} \tag{4.4}$$

if G'_1 and G'_2 are merged into $G'_{\{1,2\}}$, then to evaluate a query pattern that involves the predicates `foaf:name` and `dbo:birthPlace`, we only transfer $G'_{\{1,2\}}$ rather than $G'_1 \cup G'_2$. Note that the most important consideration is that all the subjects are a matching result for those queries only involving the predicates in $G'_{1,2}$.

Following similar premises, Gubichev and Neumann [GN14] establish a hierarchy of characteristic sets, in each step removing one element of the set and keeping only the one that minimizes the query costs (i.e. cost can be understood as cardinality, in this context). For instance, in the previous example, the approach by Gubichev and Neumann will inspect all combinations of two predicates, $F^1_{\{1,2\}}$={`foaf:name, dbp:birthdate`},

[3]Note that we consider the identity merge, i.e., $F'_{\{j\}} = F'_j$

$F^2_{\{1,2\}}$=`{foaf:name, dbp:title}`, etc., to select the one with smallest cardinality, e.g. $F^2_{\{1,2\}}$, for query planning. We use a similar idea, but the main differences with the previous work are that (i) we do not compute all predicate subsets of a given family (this was used to estimate join costs [GN14]) but only those subsets that represent merges, corresponding to non-empty intersections with other families, and (ii) we keep all these intersections in a map, irrespective of their cardinality.

To create this merged families map for all potentially non-empty intersections of sub-families, we start from $F'(G) = \{F'_1, \dots F'_m\}$, and iteratively construct a partial map μ such that, given a set of predicates f, $\mu(f)$ returns (whenever f corresponds to a non-empty intersection) a set of indexes of all original families that contain subjects contributing to f, as shown in Alg. 4.1. We initialize μ with $F'(G)$ (lines 2–4), and then, iteratively, until μ does not change anymore (lines 5–19), create mappings (corresponding to a merged family) collecting all indexes, for each non-empty intersection of families (lines 10–15). If there already is a (merged) family corresponding to the intersection found, i.e., $f \cap g$ appears already in the domain of μ (line 10), then also the corresponding index(es) are considered (line 10) and the mapping is updated, otherwise, a new mapping is created (line 14). Note that, as opposed to this pseudo-code, our actual implementation is using a hashmap for the (merged) families and avoids revisiting the same intersections repeatedly.

Then, $\mu(\cdot)$ is used to compute the partitions served by the `smart-KG` server, denoted $\mathcal{G}_{serv}$, where $\mathcal{G}'$ is replaced with a set of partitions obtained from the merged families:

$$\mathcal{G}_{serv} = \{G'_{\mu(f)} \mid f \in dom(\mu)\} \tag{4.5}$$

Note that the elements in $\mathcal{G}_{serv}$ are no longer non-overlapping, i.e., formally, they are not partitions anymore but fragments of the graph G. However, for the sake of readability, we abuse notation and refer to these fragments as *merged partitions* (or simply *partitions*). The advantage of serving these merged partitions is that the client can determine a unique minimal matching partition among $\mathcal{G}_{serv}$ to answer a query using the mapping μ as we will show in our full example in Section 4.3.1.

4.2.1.3 Family Pruning

Note that, in practice, it might be still too expensive to materialize partitions for *all* potential merges (intersections) of all families in G. For instance, as we will show in our evaluation, in the DBpedia graph, a naive merge would create +600k partially very large families, which are unfeasible to serve.

To this end, we present a family pruning strategy for further restricting the number of materialized partitions, where we (i) restrict considered predicates in P'_G based on their cardinality, (ii) avoid the creation of small families that deviate only slightly from other overlapping, "core" families, and (iii) avoid materialization of families over a certain size.

Algorithm 4.1: Family Grouping

Input : $F'(G) = \{F'_1, \dots F'_m\}$, the set of different (restricted) families.
Output : $\mu(\cdot)$ a partial mapping from sets of predicates to index sets $I \in 2^{\{1,\cdots,m\}}$

```
1  Initialize μ with the original families:
2  foreach f ∈ F'(G) do
3  |  μ(F'_i) ← {i}
4  end
5  repeat
6  |  μ'(·) ← μ(·)
7  |  foreach f ∈ dom(μ) do
8  |  |  foreach g ∈ dom(μ) do
9  |  |  |  if g ∩ f ≠ ∅ then
10 |  |  |  |  if g ∩ f ∈ dom(μ) then
11 |  |  |  |  |  μ(g ∩ f) ← μ(g ∩ f) ∪ μ(g) ∪ μ(f)
12 |  |  |  |  end
13 |  |  |  |  else
14 |  |  |  |  |  μ(g ∩ f) ← μ(g) ∪ μ(f)
15 |  |  |  |  end
16 |  |  |  end
17 |  |  end
18 |  end
19 until μ ≠ μ';
20 return μ
```

(i) Restrict predicates based on cardinality. The cardinality of predicates is a key factor in determining the number and size of shipped partitions. Therefore we distinguish between infrequent and frequent predicates in the KG.

Infrequent predicates are those that occur rarely in the KG compared to the most commonly occurring predicates. Infrequent predicates may be scattered across various subjects in the KG, leading to the creation of multiple small families. In this case, a TPF/brTPF call efficiently evaluates a single triple pattern with an infrequent predicate without the need to transfer large intermediate results (i.e. unnecessary materialization of small family partitions).

Frequent predicates can be part of almost all families such as `dbo:wikiPageExternalLink` in DBpedia leading to an undesirable increase in the size of each family, especially if they are rarely mentioned in queries.[4]

Note specifically that although `rdf:type` is a typically frequent predicate, we do not

[4]For example, `dbo:wikiPageExternalLink` appears merely 59 times in the LSQ query log, or when they remain entirely unqueried, as `dbo:wikiPageLength`, which is not mentioned in the LSQ query log.

exclude it at this point, as we will tackle this issue separately in (ii), in the handling of typed partitions.

To control predicates cardinalities, we use thresholds $\tau_{p_{low}}, \tau_{p_{high}}$ with $0 \leq \tau_{p_{low}} < \tau_{p_{high}} \leq 1$, to delimit the minimum and maximum percentage of triples per predicate, and define P'_G accordingly based on these two thresholds:

$$P'_G = \{p' \in pred(G) \mid \tau_{p_{low}} \leq \frac{|(s, p', o) \in G|}{|G|} \leq \tau_{p_{high}}\} \tag{4.6}$$

Note that publishers might still consider including particular heavy hitters (e.g. `rdf:type`) which can be frequent in queries as we present in Chapter 6.

(ii) Avoid the creation of small families. In order to address issue (ii), we aim at considering only "core" families for the partition merging process, i.e., we select predicate combinations (i.e, families) that are used by a proportionally large number of subjects, above a threshold α_s. That is, we define these core families as

$$F'_{core} = \{F'_i \in F' \mid \frac{|subj(G'_i)|}{|subj(G)|} \geq \alpha_s\} \tag{4.7}$$

with the respective index set $I_{core} = \{i \mid F'_i \in F'_{core}\}$ and predicate set $P'_{core} = \{p \in F'_i \mid F'_i \in F'_{core}\}$.

Intuitively, these *core families* represent the structured parts of the graph, i.e., star-shaped sub-graphs where entities are described with the same attributes.

(iii) Avoid the creation of large families. Finally, we avoid the materialization of overly large (e.g. hundreds of millions of triples in DBpedia) merged partitions G_I with size G_I above a threshold α_t, which limits the size of the materialized merged partitions.

In order to only take core families into account for the creation of partitions, and limit merged families to sizes below α_t, it is sufficient to modify Eq. (4.5) as follows:

$$\mathcal{G}_{serv} = \left\{ G'_{\mu(f)} \;\middle|\; \begin{array}{c} f \in dom(\mu) \wedge \\ \mu(f) \cap I_{core} \neq \emptyset \wedge \\ \sum_{i \in \mu(f)} |G'_i| \leq \alpha_t \end{array} \right\} \cup \{G_{\{i\}} | F'_i \in F'\} \tag{4.8}$$

In Eq.(4.8), line 2 addresses issue (ii)[5] and line 3 addresses issue (iii)[6]. The last part ensures that, despite pruning, the non-merged partitions of families in F' remain being served.

[5] since $subj(G'_i) \cap subj(G'_j) = \emptyset$ for all base families $F'_i, F'_j \in F'$, by construction it holds that $|subj(G'_I)| = sum_{i \in I} |subj(G'_I)|$

[6] since $|G'_I| = \sum_{i \in I} |G'_i|$

Due to these pruning steps, no longer all the partitions corresponding to families in $dom(\mu)$ will be materialized in $\mathcal{G}_{serv}$. Therefore, in practice, we define another mapping function, μ_G, that allows us to directly map families from $dom(\mu)$ to "minimal" sets of matching partitions in $\mathcal{G}_{serv}$. In practice, we compute the partitions $\mathcal{G}_{serv}$ along with μ_G in one go. That is, we build a mapping $\mu_G : dom(\mu) \mapsto 2^{2^{\{1,\ldots,m\}}}$ that maps a family f to a set of index sets $\{I_1, \ldots I_k\}$ representing (lists of) materialized matching partitions, i.e., where $\mu_G(f) = \{I \mid G'_I \in \mathcal{G}^{\prec}_{serv}(f)\}$. For $G'_{\mu(f)} \in \mathcal{G}_{serv}$, i.e., if the respective partition is materialized, then $\mu_G(f) = \{\mu(f)\}$. In this case, $\mathcal{G}^{\prec}_{serv}(f)$ is defined as: let $\mathcal{G}_{serv}(f) = \{G'_I \in \mathcal{G}_{serv} \mid f \subseteq \mu^{-1}(I)\}$ be all materialized partitions matching a family f, then $\mathcal{G}^{\prec}_{serv}(f)$ is the $\prec$-*minimal* subset of $\mathcal{G}_{serv}(f)$ with $\prec$ defined as: $G'_{I_1} \prec G'_{I_2}$ iff $\mu^{-1}(I_1) \subset \mu^{-1}(I_2)$. That is, as the partition merging can result in no longer disjoint partitions in $\mathcal{G}_{serv}$, the intuition is to pick, at query time, the partitions that are "subset-minimal with respect to their corresponding families". In practice, `smart-KG` materializes the partitions in $\mathcal{G}_{serv}$ as HDT files.

4.2.1.4 Server Operators

The `smart-KG` server materializes all partitions in $\mathcal{G}_{serv}$ into HDT files and provides operators to ship partitions and their metadata based on μ_G, or to respond to TPF requests. Overall, the following operations[7] are provided:

- *TPF(tp)* to retrieve the answer for a triple pattern tp, i.e., the `smart-KG` server returns the triples from G that match tp.
- *TPFcard(tp)* to retrieve the resulting cardinality of a triple pattern (this is a standard TPF API function).
- retrievePartition(id) to retrieve a partition by id (we use ids corresponding to partitions in $\mathcal{G}_{serv}$).
- *retrieveIDs(f)* to retrieve the IDs of $\prec$-*minimal* partitions matching a given family f (i.e., $\mu_G(f)$), plus metadata with descriptive statistics per ID (e.g, number of triples).
- *getPartitionMetadata()* to retrieve the pruning parameters used by the server (i.e., P'_{core}, τ_l, τ_h, α_s, and α_t).

As for the retrieveIDs(f) operation, it essentially scans $dom(\mu_G)$ to determine the single (cardinality-wise) smallest $f' \supseteq f$ in $dom(\mu_G)$ and retrieves IDs corresponding to the index-sets $\mu_G(f')$. Note that f' is uniquely determined, which can be proven by contradiction: i.e. assume two cardinality-wise smallest $f'_1, f'_2 \in dom(\mu_G)$ with $f'_1 \neq f'_2$ and $f \subset f'_1, f \subset f'_2$; then, also $f \subset f'_1 \cap f'_2$, where (by assumption $f'_1 \neq f'_2$) it holds that

[7] We assume that the server handles a single graph. For multiple graphs, the id of the graph can be added as a parameter.

$|f'_1 \cap f'_2| < |f'_1|$ or $|f'_1 \cap f'_2| < |f'_2|$. However, by construction of μ_G, this also implies that $f'_1 \cap f'_2 \in dom(\mu_G)$, which contradicts the assumption.

4.2.2 SMART-KG Client

The `smart-KG` client (cf. Fig. 4.3) implements partition and triple pattern shipping to efficiently execute SPARQL queries over the `smart-KG` server. The `smart-KG` client maintains a *catalog* with metadata about the families available at a `smart-KG` server obtained with the server operation *getPartitionMetadata()*. The input of the client is a SPARQL query, which the *query parser* translates into the corresponding SPARQL algebra expressions. Then, the *query decomposer* splits the BGPs within the query into star-shaped sub-queries around the same subject. Based on this decomposition, the *query optimizer* implements heuristics to determine the order of stars and triple patterns within the stars, and the shipping strategies to evaluate them. The *query executor* evaluates the plan and combines the results locally by joining the data retrieved from the server. The results produced by the engine are translated by the *results serializer* into the format specified by the user. The partitions downloaded from the `smart-KG` server during query evaluation can be stored in the *family cache* and reused for subsequent query evaluations. In the following, we will describe the main `smart-KG` client components: query decomposer, optimizer, and executor.

4.2.2.1 Query Decomposer

First, `smart-KG` splits parsed Basic Graph Patterns (BGPs) into *stars* as follows: given a BGP Q, with subjects $subj(Q)$, a decomposition $\mathcal{Q} = \{Q_s \mid s \in subj(Q)\}$ of Q is a set of star-shaped BGPs Q_s such that $Q = \bigcup_{s \in subj(Q)} Q_s$ and:

$$Q_s = \{tp \in Q \mid tp = (s, p, o)\} \tag{4.9}$$

Analogous to graphs, we can also associate a family to each Q_s:

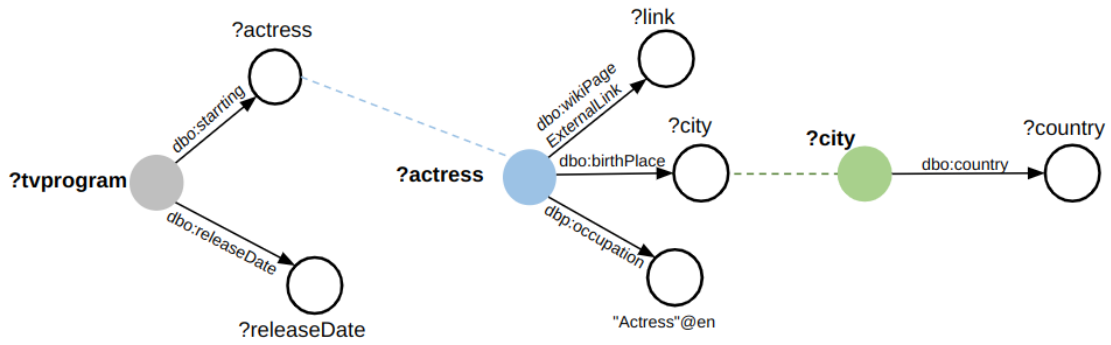
$$F(Q_s) = \{p \mid \exists o : (s, p, o) \in Q_s, p \in U\} \tag{4.10}$$

Given the SPARQL query in Fig. 4.4a, the BGP is decomposed into $\mathcal{Q} = \{Q_{\texttt{?tvprogram}}, Q_{\texttt{?actress}}, Q_{\texttt{?city}}\}$ around the three subjects (cf. Fig. 4.4b). Each of the star families $F(Q_s)$ that can be mapped to existing predicate families in $dom(\mu_G)$ on the server has a non-empty answer. For example, $Q_{\texttt{?tvprogram}} = \{$(`?tvprogram`, `dbo:starring`, `?actress`), (`?tvprogram`, `dbo:releaseDate`, `?releaseDate`)$\}$ has $F(Q_{\texttt{?tvprogram}}) =$ {`dbo:starring`, `dbo:releaseDate`};

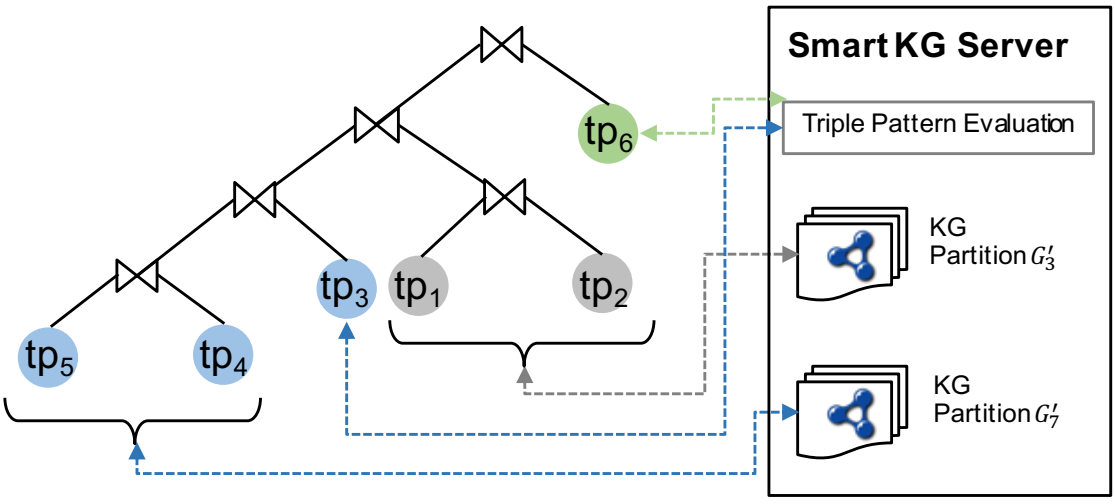
based on the decomposition $\mathcal{Q}$, the `smart-KG` client's shipping-based query optimizer next has to devise a query plan.

```
SELECT * WHERE {
    ?tvprogram dbo:starring ?actress .              # tp1  544,110 matches
    ?tvprogram dbo:releaseDate ?releaseDate .       # tp2  155,199 matches
    ?actress dbo:wikiPageExternalLink ?link .       # tp3  9,643,439 matches
    ?actress dbo:birthPlace ?city .                 # tp4  1,469,160 matches
    ?actress dbp:occupation "Actress"@en.           # tp5  18,861 matches
    ?city dbo:country ?country.                     # tp   789,261 matches
}
```

(a) Select all actresses, their TV programs, and birthplace information



(b) Star-shaped query decomposition



(c) Shipping plan based on the decomposition

Figure 4.4: Example of processing a SPARQL query with the `smart-KG` client.

4.2.2.2 Shipping-based Query Planner & Optimizer

The `smart-KG` client query planner devises plans where both triple pattern results (using TPF) and partitions in $\mathcal{G}_{serv}$ are transferred from the server to resolve the sub-queries in $\mathcal{Q}$. To decide whether and for which sub-queries to use triple pattern or partition shipping, and in which order to execute them, the optimizer implements heuristics based on the sub-queries in $\mathcal{Q}$ and the server's partition metadata.

The resolution of each sub-query can be then performed in two ways in `smart-KG`: (i) locating the appropriate family(ies) on the `smart-KG` server, shipping the corresponding HDT partition, and performing the sub-query locally in the client, or (ii) resolving the sub-query using TPF and shipping sub-query results, doing all join operations locally.

Partition Shipping (P-S). Shipping relevant partitions to evaluate a star $Q_s \in \mathcal{Q}$ needs to take into consideration the materialized partitions at the server. Since graph partitions are generated based on the core families (cf. Section 4.2.1), only stars with $F(Q_s) \subseteq P'_{core}$ can be fully evaluated by served partitions. Therefore, the optimizer first partitions each $Q_s \in \mathcal{Q}$ into the disjoint sets Q'_s and Q''_s, where $Q'_s = \{(s,p,o) \in Q_s \mid p \in P'_{core}\}$, i.e., the part of the star that can be evaluated over the served partitions, whereas the remaining triple patterns in $Q''_s = Q_s \setminus Q'_s$ are delegated to TPF requests.[8]

Then, the optimizer implements the following additional heuristics: partition shipping is only followed if $|Q'_s| > 1$, as in practice, the transfer of graph partitions to resolve a single triple pattern usually takes longer than delegating to a TPF request directly.

Triple Pattern Shipping (TP-S). Triple patterns *tp* delegated to TPF will be evaluated using a *TPF(tp)* request to the server. This involves the triples patterns in Q''_s and Q'_s with $|Q'_s| = 1$ in addition to triple patterns with variables in the predicate position, as these cannot be associated with family a partition directly.

The query optimizer, given $\mathcal{Q}$ and P'_G as input, devises a query plan Π_Q, based on the described sub-decomposition into P-S and TP-S patterns. It accordingly proceeds in two phases, first iterating over each star $Q_s \in \mathcal{Q}$ to perform the partitioning into Q'_s and Q''_s, additionally collecting the cardinality for each triple pattern $tp_i \in Q_s$ using *TPFcard()* server requests. Then, the optimizer devises sub-plans, for Q'_s and Q''_s that can be efficiently executed, by join ordering based on these cardinalities, using the construct *Plan*, that comprises a pair of a left-linearly ordered query plan, along with a shipping strategy. Join order is determined by the cardinality of triple patterns, where smaller triple patterns are evaluated first. For each sub-query Q_s, the optimizer creates a shipping-based sub-plan Π_s which is added to the set of current *subplans*.

Fig. 4.4c shows the shipping strategies for each sub-plan from our example. For the sub-query $Q_{\texttt{?actress}}$, the optimizer created $Plan((tp_5 \bowtie tp_4), P\text{-}S)$. Yet, the triple pattern tp_6 in $S_{\texttt{?actress}}$ is evaluated using triple pattern shipping as the optimizer determined that the predicate `dbo:wikiPageExternalLink` is not in P'_G.

[8] Note that Q''_s also includes triple patterns with predicate variables, i.e., $p \in V$.

Algorithm 4.2: Query Executor: *evalPlan*

```
   Input: Query plan Π
   Output: Ω the result set of executing Π
 1 if Π = Plan(Π_s, P-S) then
 2 |   Q_s is the sub-query associated with Π_s
 3 |   G* = {getPartition(id) | id ∈ retrieveIds(F(Q_s))}
 4 |   Ω ← {ω_∅}
 5 |   for tp_i ∈ P_s do
 6 |   |   Ω ← Ω ⋈ ⋃_{G_j ∈ G*} [[tp_i]]_{G_j}
 7 |   end
 8 end
 9 else if Π = Plan(Π_s, TP-S) then
10 |   Ω ← TPF(Π_s)
11 end
12 else
13 |   Π is (Π_l ⋈ Π_r)
14 |   Ω ← evalPlan(Π_l) ⋈ evalPlan(Π_r)
15 end
16 return Ω
```

In the second phase, the optimizer combines the sub-plans that share variables to build the final plan Π_Q. Again, the optimizer uses a heuristic to determine the join order based on selecting the sub-plan Π_i containing the overall smallest (i.e., assumed most selective) triple pattern from *subplans* first, and so on, iteratively joining sub-plans to Π_Q. The resulting query plan Π_Q comprises sub-plans annotated with the corresponding shipping strategy, and join operators to be evaluated locally by the client.

4.2.2.3 Query Executor

The function *evalPlan* evaluates the plan Π_Q by traversing the tree of sub-plans (cf. Alg. 4.2). The shipping strategies are implemented by calling the respective `smart-KG` server operators (cf. Sec. 4.2.1.4). Depending on the structure of the sub-plans, the query executor distinguishes the following cases.

Case: (P-S) Sub-plans. P-S sub-plans are evaluated (cf. Alg. 4.2, lines 1–6) by determining relevant served partition IDs for Q'_s, through calling $retrieveIds(F(Q'_s))$, and retrieving each ID from the server (line 3). The query executor evaluates the triple patterns tp_i against each such partition and merges the results using the SPARQL algebra union operator (line 4). The intermediate results of each triple pattern are joined in following the plan Π_s (lines 5–6).

Case: (TP-S) Sub-plans. TP-S sub-plans are composed of single triple patterns, executable by calling the *TPF(tp)* `smart-KG` server operator (cf. Alg. 4.2, lines 7–8).

General Case. Joins the results of two recursively evaluated sub-plans Π_l and Π_r (cf. Alg. 4.2, lines 9–11).

The outcome of the query executor is the result set Ω of evaluating the query Q. In practice, the executor implements an iterator model to push intermediate results of

evaluating one subplan to the next operator in the plan. This allows the `smart-KG` client for streaming results incrementally as the data arrives from the server.

Proposition 1. *The result of evaluating a BGP Q over an RDF graph G with `smart-KG`, denoted smart-eval(Q, G), is correct w.r.t. the semantics of the SPARQL language, i.e., smart-eval$(Q, G) = [\![Q]\!]_G$.*[9]

4.3 Proof of `smart-KG` Correctness

Proof. For this proof, we assume that the server operators are implemented correctly. By contradiction, let us assume that *smart-eval*$(Q, G) \neq [\![Q]\!]_G$. We distinguish three cases based on the shipping strategy used for evaluating Q.

(i) Q is evaluated with P-S. For this case, we assume the correct implementation of the join operator, therefore, it is sufficient to prove this case when Q is composed of a single triple pattern with free-variable-predicate p. With P-S, the evaluation of Q is carried out against the set of corresponding partitions. By definition of the server operators, the ids of the relevant partitions for the predicate family of Q is equivalent to $\mu_G(F(Q))$. Furthermore, after applying the server operators, we obtain that the set of relevant partitions G^* for Q is $G^* = \{G_j \in \mathcal{G}_{serv} \mid j \in \mu(F(Q))\}$. Next, we consider two sub-cases. In the first sub-case, we have that *smart-eval*$(Q, G) \subset [\![Q]\!]_G$, i.e., there exists an RDF triple $t \in G$ with predicate p such that $t \notin \bigcup_{G_j \in G^*} G_j$. Therefore, the partitions in $\mathcal{G}_{serv}$ are created incorrectly, which contradicts Equation 4.8. The sub-case $[\![Q]\!]_G \subset$ *smart-eval*(Q, G) does not occur as, by definition of partitions, every $G_j \in G^*$ is a set of RDF triples with predicate p.

(ii) Q is evaluated with TP-S. For this case, the evaluation of Q is carried out as *TPF*(Q) and Q corresponds to a single triple pattern (which is ensured by the query optimizer). By hypothesis, *TPF*(Q) does not produce $[\![Q]\!]_G$, which contradicts the definition of the *TPF* server operator.

(iii) Q is evaluated following a Hybrid Shipping Strategy with TP-S and P-S. This case follows immediately from cases (i) and (ii) because any hybrid strategy involving TP-S and P-S would still be affected by the contradictions presented in cases (i) and (ii). □

4.3.1 Detailed Example:

In this section, we demonstrate a full example to explain the evaluation of the SPARQL query based on our introduced approach `smart-KG`. In this example, we elaborate family-based partitioning and SPARQL query evaluation on our RDF graph from Example 7 inspired *Friends* series:

[9] Experiments and the proof are available online `https://ai.wu.ac.at/smartkg`

Creation of Family-based partitioning for Friends RDF graph. In this example, we materialize family-based partitions based on three different settings, as follows:

- Setting 1: we materialize all families as well as all possible merges of families (i.e. the pruning step is not applied). In our example, it is feasible to materialize all partitions given the fact that our example graph is small of size $|G| = 79$ triples and $|P_G| = 16$ predicates. For this purpose, we fix the pruning parameters as follows: we set $\tau_l = 0$ and $\tau_h = 1.0$ so that we include all predicates including both infrequent and heavy hitters. Likewise, we set $\alpha_s = 0$ to include all families in the grouping step and $\alpha_t = 1.0$ to materialize all possible family merges.

  This setting will generate $|\mathcal{G}_{serv}| = 18$ materialized partitions based on the following set of families:

  - F_1={`dbo:portrayer`, `dbp:children`, `dbp:family`, `dbp:gender`, `dbp:occupation`, `dbp:religion`, `rdf:type`, `rdfs:label`} which is an *original* family that generates a partition G_1 where $|G_1| = 9$ triples with a set of subjects $subj(G_1)$ of size $|subj(G_1)| = 1$ subjects. This subject represents the fictional Chandler since she is the only member of the graph that have the combination of predicates in F_1.
  - F_2={`dbo:birthDate`, `dbo:education`, `dbp:occupation`, `dbp:spouse`, `rdf:type`, `rdfs:label`} which is an *original* family that generates a partition G_2 where $|G_2| = 18$ triples with a set of subjects $subj(G_2)$ of size $|subj(G_2)| = 3$ subjects. These subjects represent two actresses Courteney Cox and Jennifer Aniston and one actor David Schwimmer as they share the set of predicates in F_2.
  - F_3={`dbo:almaMater`, `dbo:birthDate`, `dbp:occupation`, `dbo:spouse`, `rdf:type`, `rdfs:label`} which is an *original* family that generates a partition G_3 where $|G_3| = 6$ triples with a set of $subj(G_3)$ of size $|subj(G_3)| = 1$ subject. This subject represents the actress Lisa Kudrow.
  - F_4={`dbo:birthDate`, `dbp:occupation`, `dbp:spouse`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_4 where $|G_4| = 30$ triples with a set of $subj(G_4)$ of size $|subj(G_4)| = 7$ subjects. These subjects represent the two actors Matt LeBlanc and David Schwimmer as well as three actresses Courteney Cox, Jennifer Aniston, and Lisa Kudrow in addition to the fictional character Ross Geller as they share the set of predicates in F_4.
  - F_5={`dbo:birthDate`, `dbp:occupation`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_5 where $|G_5| = 28$ triples with a set of $subj(G_5)$ of size $|subj(G_5)| = 7$ subjects. These subjects represent the two actors Matthew Perry and David Schwimmer as well as three actresses Courteney Cox, Jennifer Aniston, and Lisa Kudrow in addition to the fictional character Ross Geller as they share the set of predicates in F_5.

- F_6={dbo:portrayer, dbp:occupation, rdf:type, rdfs:label} which is a *grouped* family that generates a partition G_6 where $|G_6| = 25$ triples with a set of $subj(G_6)$ of size $|subj(G_6)| = 6$ subjects. These subjects actually represent the 6 fictional characters of the show including Ross, Monica, Chandler, Joey, Phoebe, and Rachel as they share the set of predicates in F_6.
- F_7={dbo:alias, dbo:portrayer, dbo:affiliation, dbp:nationality, dbp:occupation, dbp:religion, rdf:type, rdfs:label} which is an *original* family that generates a partition G_7 where $|G_7| = 8$ triples with a set of $subj(G_7)$ of size $|subj(G_7)| = 1$ subject. This subject represents the fictional character Joey Tribbiani.
- F_8={dbo:alias, dbo:portrayer, dbp:children, dbp:occupation, dbp:religion, dbp:spouse, rdf:type, rdfs:label} which is an *original* family that generates a partition G_8 where $|G_8| = 8$ triples with a set of $subj(G_8)$ of size $|subj(G_8)| = 1$ subject. This subject represents the fictional character Monica Geller.
- F_9={dbo:portrayer, dbp:family, dbp:gender, rdf:type, dbp:nationality, dbp:occupation, rdfs:label} which is an *original* family that generates a partition G_9 where $|G_9| = 8$ triples with a set of $subj(G_9)$ of size $|subj(G_9)| = 1$ subject. This subject represents the fictional character Phoebe Buffay.
- F_{10}={dbo:birthDate, dbo:portrayer, dbp:children, dbp:occupation, dbp:religion, dbp:spouse, dbp:title, rdf:type, rdfs:label} which is an *original* family that generates a partition G_{10} where $|G_{10}| = 9$ triples with a set of $subj(G_{10})$ of size $|subj(G_{10})| = 1$ subject. This subject represents the fictional character Ross Geller.
- F_{11}={dbp:occupation, rdf:type, rdfs:label} which is a *grouped* family that generates a partition G_{11} where $|G_{11}| = 37$ triples with a set of $subj(G_{11})$ of size $|subj(G_{11})| = 6$ subject. These are all unique subjects in the Friends graph as all actors, actresses, and fictional characters have the set of predicates in F_{11}.
- F_{12}={dbo:portrayer, dbp:occupation, dbp:religion, rdf:type, rdfs:label} which is a *grouped* family that generates a partition G_{12} where $|G_{12}| = 20$ triples with a set of $subj(G_{12})$ of size $|subj(G_{12})| = 4$ subjects. These subjects actually represent the 4 fictional characters of the show including Ross, Monica, Chandler, and Joey as they share the set of predicates in F_{12}.
- F_{13}={dbo:portrayer, dbp:children, dbp:religion, dbp:occupation, rdf:type, rdfs:label} which is a *grouped* family that generates a partition G_{13} where $|G_{13}| = 18$ triples with a set of $subj(G_{13})$ of size $|subj(G_{13})| = 3$ subjects. These subjects actually represent the 3 fictional characters of the show including Ross, Monica, and Chandler as they share the set of predicates in F_{13}.

- F_{14}={dbo:portrayer, dbp:family, dbp:gender, dbp:occupation, rdf:type, rdfs:label}
  which is a *grouped* family that generates a partition G_1 where $|G_{14}| = 14$ triples with a set of $subj(G_{14})$ of size $|subj(G_{14})| = 2$ subjects. These subjects actually represent the 2 fictional characters Chandler and Phoebe as they share the set of predicates in F_{14}.
- F_{15}={dbp:occupation, dbp:spouse, rdf:type, rdfs:label}
  which is a *grouped* family that generates a partition G_{15} where $|G_{15}| = 28$ triples with a set of $subj(G_{15})$ of size $|subj(G_{15})| = 7$ subjects. These subjects represent the two actors Matt LeBlanc and David Schwimmer as well as three actresses Courteney Cox, Jennifer Aniston, and Lisa Kudrow in addition to the fictional character Ross Geller as they share the set of predicates in F_{15}.
- F_{16}={dbo:alias, dbo:portrayer, dbp:occupation, dbp:religion, rdf:type, rdfs:label} which is a *grouped* family that generates a partition G_{16} where $|G_{16}| = 12$ triples with a set of $subj(G_{16})$ of size $|subj(G_{16})| = 2$ subjects. These subjects represent the two fictional characters Joey and Monica.
- F_{17}={dbo:portrayer, dbp:nationality, dbp:occupation, rdf:type, rdfs:label} which is a *grouped* family that generates a partition G_{17} where $|G_{17}| = 10$ triples with a set of $subj(G_{17})$ of size $|subj(G_{17})| = 2$ subjects. These subjects represent the two fictional characters Joey and Phoebe.
- F_{18}={dbo:portrayer, dbp:children, dbp:occupation, dbp:religion, dbp:spouse, rdf:type, rdfs:label} which is a *grouped* family that generates a partition G_{18} where $|G_{18}| = 14$ triples with a set of $subj(G_{18})$ of size $|subj(G_{18})| = 2$ subjects. These subjects represent the two fictional characters Monica and Ross.

In Setting 1, we materialize all possible partitions $G_{serv} = \{G_1, ..., G_{18}\}$.

- Setting 2: we materialize family-based partitions based on our introduced family pruning strategy. We fix $\tau_l = 3/100$ and $\tau_h = 20/100$ which allow $|P'_G| = 13$ predicates and prune the following infrequent predicates = {dbp:title, dbo:almaMater, dbp:affiliation}. We set $\alpha_t = 0.05$ and $\alpha_s = 0$. These settings restrict the generated families to the following:
  - F_1={dbo:portrayer, dbp:children, dbp:family, dbp:gender, dbp:occupation, dbp:religion, rdf:type, rdfs:label}
    which is an *original* family that generates a partition G_1 where $|G_1| = 9$ triples with a set of subjects $subj(G_1)$ of size $|subj(G_1)| = 1$ subjects. This subject represents the fictional Chandler since she is the only member of the graph that have the combination of predicates in F_1.

- F_2={dbo:birthDate, dbo:education, dbp:occupation, dbp:spouse, rdf:type, rdfs:label}
  which is an *original* family that generates a partition G_2 where $|G_2| = 18$ triples with a set of subjects $subj(G_2)$ of size $|subj(G_2)| = 3$ subjects. These subjects represent two actresses Courteney Cox and Jennifer Aniston and one actor David Schwimmer as they share the set of predicates in F_2.
- F_{3+4}[10]={dbo:birthDate, dbp:occupation, dbp:spouse, rdf:type, rdfs:label}
  which is an *original* family that generates a partition G_{3+4} where $|G_{3+4}| = 110$ triples with a set of $subj(G_{3+4})$ of size $|subj(G_{3+4})| = 2$ subjects. These subjects represent the actors Lisa Kudrow and Matt LeBlanc. On pruning the predicate dbo:almaMater, both Lisa and Matt have identical predicates which generates F_{3+4}. However, we did not materialize the grouped version of the family (similar to F_4 in Setting 1 since it exceeds the threshold $\alpha_t = 0.05$ that we adopt here in Setting 2.
- F_5={dbo:birthDate, dbp:occupation, rdf:type, rdfs:label}
  which is an *original* family that generates a partition G_5 where $|G_5| = 4$ triples with a set of $subj(G_5)$ of size $|subj(G_5)| = 1$ subjects. This single subject represents Matthew Perry which is the only subject with this set of predicates.
- F_6={dbo:portrayer, dbp:occupation, rdf:type, rdfs:label}
  which is an *original* family that generates a partition G_6 where $|G_6| = 4$ triples with a set of $subj(G_6)$ of size $|subj(G_6)| = 1$ subjects. This single subject represents Rachel Green which is the only subject with this set of predicates.
- F_7={dbo:alias, dbo:portrayer, dbp:nationality, dbp:occupation, dbp:religion, rdf:type, rdfs:label}
  which is an *original* family that generates a partition G_7 where $|G_7| = 7$ triples with a set of $subj(G_7)$ of size $|subj(G_7)| = 1$ subject. This subject represents the fictional character Joey Tribbiani.
- F_8={dbo:alias, dbo:portrayer, dbp:children, dbp:occupation, dbp:religion, dbp:spouse, rdf:type, rdfs:label}
  which is an *original* family that generates a partition G_8 where $|G_8| = 8$ triples with a set of $subj(G_8)$ of size $|subj(G_8)| = 1$ subject. This subject represents the fictional character Monica Geller.
- F_9={dbo:portrayer, dbp:family, dbp:gender, dbp:nationality, dbp:occupation, rdf:type, rdfs:label}
  which is an *original* family that generates a partition G_9 where $|G_9| = 8$ triples with a set of $subj(G_9)$ of size $|subj(G_9)| = 1$ subject. This subject represents the fictional character Phoebe Buffay.

[10]For illustration purposes, the naming convention F_{3+4} is employed to maintain consistency with the identifiers of the family in other settings, facilitating ease of comprehension for the reader.

– F_{10}={dbo:birthDate, dbo:portrayer, dbp:children, dbp:occupation, dbp:religion, dbp:spouse, rdf:type, rdfs:label}
which is an *original* family that generates a partition G_{10} where $|G_{10}| = 8$ triples with a set of $subj(G_{10})$ of size $|subj(G_{10})| = 1$ subject. This subject represents the fictional character Ross Geller.

In Setting 2, $G_{serv} = \{G_1, ..., G_9\}$ with materialized partitions from original families, excluding infrequent predicates. In this setting, merged families have not been materialized due to partitions size exceeding our predetermined threshold $\alpha_t = 0.05$ of the graph size which limits the size of the materialized merged families.

- Setting 3: we slightly relax the threshold of $\alpha_t = 0.05$ to materialize more grouped families as we set $\alpha_t = 0.30$. Similar to Setting 2, we fix $\tau_l = 3/100$, $\tau_h = 20/100$, and $\alpha_s = 0$ which allow $|P'_G| = 13$ predicates and prune the following infrequent predicates = {dbp:title, dbo:almaMater, dbp:affiliation}. These settings restrict the generated families to the following:

  – Original Families: we materialize the following original families including F_1, F_2, F_{3+4}, F_5, F_6, F_7, F_8, and F_9, F_{10} similar to Setting 2.
  – Grouped Families: we materialize the following grouped families including F_{12}, F_{13}, F_{14}, F_{16}, F_{17}, and F_{18}.

  In Setting 3, $G_{serv} = \{G_1, G_2, F_{3+4}, ...G_9, G_{11}, ..., G_{14}, G_{16}, ..., G_{18}\}$. We prune the partitions that can be generated from the following merged families F_4, F_{11}, F_{15} (see Setting 1) as these partitions exceed the predetermined threshold $\alpha_t = 0.30$ of the graph size.

SPARQL Query Evaluations on Friends RDF graph. In the following, we show the evaluation of SPARQL query examples using the materialized partitions according to the aforementioned settings:

Query 1: retrieve the list of characters featured in TV shows, including their respective occupations and information regarding the actors who portray them. Specifically, retrieve the birthdate and actual educational background of the actors in real life. This query Q can be written, as follows:

```
SELECT * WHERE {
    ?character dbo:portrayer ?portrayer.
    ?character dbo:occupation ?occupation.
    ?character rdf:type ?type.
    ?portrayer dbo:birthDate ?date.
    ?portrayer dbo:education ?education.
}
```

First, the query decomposer splits the BGP into two star-shaped sub-queries as follows: $\mathcal{Q} = \{Q_{?character}, Q_{?portrayer}\}$. Second, `smart-KG` query planner devises a plan where both of the decomposed stars can be fully evaluated by the served partitions since $F(Q_{?character}) \subseteq P'_{core}$ and $F(Q_{?portrayer}) \subseteq P'_{core}$ in all of the aforementioned partitioning settings. This left-linear plan is written as the following:

$\Pi = \{Plan(\Pi_{?portrayer}, \text{P-S}), Plan(\Pi_{?character}, \text{P-S})\}$.

In the following, we detail the evaluation of the query plan Π using the query executor(cf. Algo. 4.2), while considering the different partitioning settings:

- Case Setting 1: the query executor starts with evaluating $Plan(\Pi_{?portrayer}, \text{P-S})$ by shipping the partition G_2 based on the original family F_2. Note that F_2 is only family that contains the set of predicates of $Q_{?portrayer}$ in G_{serv}. The size of the intermediate results of evaluating $Q_{?portrayer}$ triple patterns on G_2 is $|\Omega_{?portrayer}| = 18$. Then, The query executor evaluates $Plan(\Pi_{?character}, \text{P-S})$ by determining the relevant served partition for $\Pi_{?character}$ (line 3) which is a single partition G_6 based on the grouped family F_6 to resolve the star query $Q_{?character}$. The query executor evaluates each triple pattern tp_i in $Q_{?character}$ on G_6 (line 6) generating the following intermediate results $\Omega_{?character}$ that is of a size equal to $|\Omega_{?character}| = |G_6| = 25$. Note that F_1, F_6, F_7, F_8, F_9, F_{10}, F_{12}, F_{13}, F_{14}, F_{17}, and F_{18} include all the predicates from $Q_{?character}$. The families F_1, F_7, F_8, F_9, and F_{10} are original families, while F_6, F_{12}, F_{13}, F_{14}, G_{17}, and F_{18} are grouped families and we ship, if exists, a grouped family. To ensure that the complete solution is contained within the shipped partition(s), we select the subset-minimal (grouped) family which is in this case F_6. Finally, we join the solution mappings of the two star sub-queries to compute the final solution mapping (line 14), as $\Omega_Q = \Omega_{?portrayer} \bowtie \Omega_{?character}$ and the result is in the following:

  $\textsc{Select}_*(\Omega_Q) = \{\{character \rightarrow$ dbr:Rachel_Green,
  $portrayer \rightarrow$ dbr:Jennifer_Aniston, $occupation \rightarrow$ dbr:Louis_Vuitton,
  $type \rightarrow$ dbo:FictionalCharacter, $date \rightarrow$ "1969 − 02 − 11",
  $education \rightarrow$ dbr:Fiorello_H._LaGuardia_High_School$\}$,
  $\{character \rightarrow$ dbr:Monica_Geller, $portrayer \rightarrow$ dbr:Courteney_Cox,
  $occupation \rightarrow$ dbr:Chef, $type \rightarrow$ dbo:FictionalCharacter,
  $date \rightarrow$ "1964−06−15", $education \rightarrow$ dbr:Mount_Vernon_Seminary_and_College$\}$,
  $\{character \rightarrow$ dbr:Ross:Geller, $portrayer \rightarrow$ dbr:David_Schwimmer,
  $occupation \rightarrow$ dbr:Chef, $type \rightarrow$ dbo:FictionalCharacter,
  $date \rightarrow$ "1966 − 11 − 02", $education \rightarrow$ dbr:Northwestern_University$\}\}$.

- Case Setting 2: The query executor starts with evaluating $Plan(\Pi_{?portrayer}, \text{P-S})$ by shipping the partition G_2 created from the original families G_2 covering $F(Q_{portrayer})$. The size of the intermediate results of evaluating $Q_{?portrayer}$ triple patterns on G_2 is $|\Omega_{?portrayer}| = 18$. Then, the query executor evaluates $Plan(\Pi_{?character}, \text{P-S})$ by requesting the relevant partitions G_1, G_6, G_7, G_8, G_9, and G_{10} generated based

the original families F_1, F_6, F_7, F_8, F_9, and F_{10} as the minimal subset of partitions required to resolve $Q_{?character}$ as in the case of not materializing grouped partitions, we ship the original families that cover $F(Q_{?character})$. This subset of partitions are locally queried on the client-side to retrieve the solution mappings of $Q_{?character}$. The size of the intermediate results is $|\Omega_{?character}| = 24$. The example demonstrates the trade-off between materializing grouped families on a hard disk and shipping original families with additional data transfer and join operations. In Setting 1, the resolution of the query $Q_{?character}$ required the shipment of 24 triples, whereas in Setting 2, 36 triples were shipped. This highlights the impact of grouping strategy on the total number of shipped triples. Finally, the final solution mappings are computed similar to Case Setting 1.

- Case Setting 3: Similar to the previous case of Setting 2, we serve G_2 to resolve $Q_{?portrayer}$. Additionally, we resolve the $Q_{?portrayer}$ by serving the partitions G_1, G_6, G_7, G_8, G_9, and G_{10} that are materialized from original families. It is worth noting that partitions from grouped families such as F_{12}, F_{14}, and F_{17} are not shipped, despite containing of the necessary predicates since the generation of a minimal subset is not possible and this will cause shipping repeated triples that belong to multiple groups would result in added cost for shipping and client-side filtering. In this case, if we decided to ship F_{12}, F_{14}, and F_{17}, we will have repeated triples representing Joey and Chandler and Phoebe while we are missing the triples representing Rachel.

4.4 smart-KG as an LDF interface (SKG)

We will give an abbreviated name to smart-KG as SKG [AFA+20] for simplicity. The server holds (compressed and queryable) partitions per common predicate families of G. As we discussed, SKG uses families as a basis for inducing a graph partitioning of G, with one partition G_f per $f \in F(G)$ [AFA+20].

We can interpret SKG as an LDF interface as follows:

- admissible patterns are defined by submitting a predicate family $f' = \{p_1, \dots p_k\}$, which may be interpreted as a pattern $\bigcup_{i=1}^{k}\{(?S, p_i, ?P_i)\}$, or resp., in SPARQL syntax, as `{ ?S` p_1 `?P1; ` p_2 `?P2; ...; ` p_k `?Pk. }`,
- $\Omega = \{\emptyset\}$ is the only admissible binding set, i.e., SKG does not consider binding restrictions,
- the selector function may be viewed as a variation of $s(\cdot)$ as follows: while the SKG server API returns a graph G_f per family $f \in F(G)$ matching P, the union of all these graphs is defined as

$$s_{\text{SKG}}(G, P, \Omega) = \{t \in G \mid \exists t' \in s(G, P, \Omega) : subj(t) = subj(t')\}$$

That is, while strictly speaking, indeed rather *several* partitions G_f are returned, $s_{\text{SKG}}(G, P, \Omega) = \bigcup_{f \supseteq f'} G_f$ defines the union of *all* these partitions $G_f \subseteq G$ such that $f' \subseteq f$ which are sent to the client,

- Φ: only $n = \infty$ is admissible, i.e., no paging is supported since the union of all relevant partitions is returned – unlike SPF an over-estimation representing all *subgraphs relevant to a star-shaped subquery*

An SKG client hence decomposes BGPs into families f' of star-shaped subqueries – on an abstract level, discarding variables or concrete bindings – and fetches via this API the subgraphs G_f (that are available in compressed form on the server) matching f'; single non-star triples in the BGP are retrieved via TPF and joins between star-shaped subqueries, and single triple queries are then computed on the client-side. As for Φ, note that it would not make sense to decompose family-based partitions into chunks since chunking up the HDT-compressed partitions would require decompression.

4.5 Experimental Evaluation

We compare the performance of `smart-KG` with state-of-the-art SPARQL engines. All datasets, queries, and results of our experiments using different workloads are available online.[9]

Knowledge Graphs. We use four RDF graphs (cf. Table 4.1): three synthetic datasets from the Waterloo SPARQL Diversity Benchmark (WatDiv) [AHÖD14], with sizes of 10M, 100M, and 1000M triples; plus, we use the real-world DBpedia [LIJ+15] dataset (v.2015A).

Table 4.1: Characteristics of the evaluated knowledge graphs

RDF Graph G	#Trip. $\lvert G\rvert$	#Subj. $\lvert S_G\rvert$	#Pred. $\lvert P_G\rvert$	#Obj. $\lvert O_G\rvert$	$\lvert P'_G\rvert$	$\lvert P'_{core}\rvert$	$\lvert F'_{core}\rvert$	$\lvert \mathcal{G}_{serv}\rvert$	C.Time (h)
WatDiv-10M	10,916,457	521,585	86	1,005,832	59	59	10,106	21,210	1
WatDiv-100M	108,997,714	5,212,385	86	9,753,266	59	59	22,855	37,392	7
WatDiv-1B	1,092,155,948	52,120,385	86	92,220,397	59	59	39,046	52,885	12
DBpedia	837,257,959	113,986,155	60,264	221,623,898	218	84	35	29,965	23

Queries and Workloads. For WatDiv, we consider 80 workloads (one per client), each of them with 154 SPARQL queries that were selected uniformly at random from the WatDiv stress test queries [AHÖD14].

The queries contain up to 10 joins with varying selectivity and shapes (star, path, and snowflake). All workloads follow nearly the same distribution of query selectivities and shapes. For DBpedia, we use queries from the real-world LSQ query log [SAH+15]; here, we are interested in highly-demanding queries, hence, we randomly selected 12 BGP queries (out of 259) with runtime higher than 1s. We report the average measures of three independent executions.

Compared Systems. We evaluate the following systems:

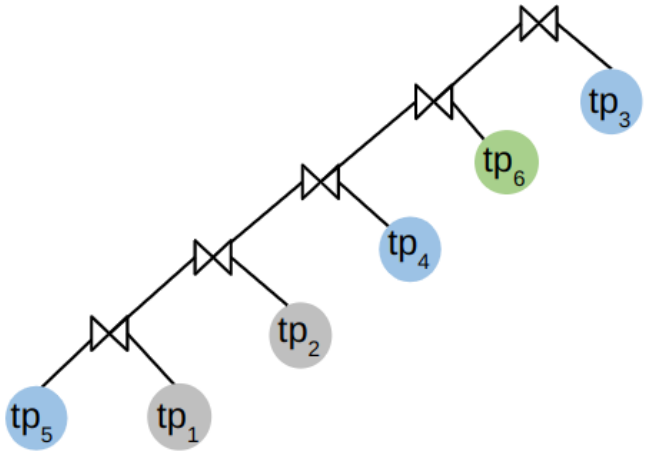


Figure 4.5: The query plan of Example 4.4a according to TPF implementation that we have followed in our Experiments

- **`smart-KG`:** We implement both client and server in Java[9], extending the TPF implementations[11]. HDT indexes and data are stored on the server's disk, with no client-side family caching. In this experimental evaluation, we have employed a query planner identical to that utilized in the Triple Pattern Fragments (TPF) implementation, which adheres to a left-linear plan at the level of triple patterns, as depicted in Figure 4.5. This plan, as illustrated in Figure 4.5, differs from the plan utilized throughout the remainder of the thesis, as we sought to ensure fair comparability with TPF and eliminate any potential performance gains derived from the query planning. It should be noted that both query plan 4.4c and 4.5 have their advantages depending on the input query. In Section 6.5, we provide a detailed evaluation of both query planners to evaluate their performance and analyze their behavior under different scenarios.
- **Triple Pattern Fragments (TPF):** We use the node.js TPF client, recommended by the authors, plus the Java TPF server [12], as the `smart-KG` TPF handler is also implemented in Java.
- **Virtuoso:** We run Virtuoso [EM09] (v7.2.5), without quotas or limits.
- **SaGe:** We use the Python SaGe server and the Java SaGe client with recommended configurations[13].

We configured Virtuoso and SaGe to run with 4 workers [MSM19]. We omit the comparison with brTPF [HA16], as existing evaluations report that the performance lie between TPF and SaGe [MSM19], and our tests show scalability problems of the brTPF server implementation [HA16] for the WatDiv-100M and Watdiv-1000M graphs.

[11] `http://linkeddatafragments.org/software/`
[12] `https://linkeddatafragments.org/software/`
[13] `https://github.com/sage-org/`

Hardware Setup. We use the following technical infrastructure.

- **Client specifications:** Clients ran on 1, 20 10, 40, and 80 physical machines, each with identical hardware specifications: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz, 32GB of DDR4 RAM, 512GB M.2 NVMe SSD, running Fedora 29 (Linux Kernel v 5.0.14).
- **Server specifications:** The servers run on a VM hosted on a machine running QEMU+ KVM hypervisor with Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz, 384 GB of DDR3 RAM, running Centos 7 (Linux Kernel v3.10.0). Compared server systems were running on VMs configured with 4 CPUs and 32 GB RAM.
- **Network configuration:** For emulating realistic internet connection bandwidth from consumer internet service providers, we limited the network speed of each client to 20 MBit, using `tc` [A+99].

Metrics. Our evaluation considers the following metrics:

- **Number of Timeouts:** Number of queries that time out. We set a timeout of 5 min. for WatDiv and 30 min. for DBpedia queries.
- **Execution Time:** Elapsed time spent by a client executing a workload, measured with the `time` command of Linux.
- **Resource Consumption:** We report on CPU usage per core, RAM usage, and network traffic. The clients and the servers were monitored using `psutil`[14], a Python library for retrieving information on running processes and system utilization.

4.5.1 Creation of Family-based Partitions

For each graph G, Table 4.2 shows the number of restricted and core predicates ($|P'_G|$, $|P'_{core}|$), core families, $|F'_{core}|$, and the materialized partitions after grouping/pruning, $|\mathcal{G}_{serv}|$, as well as the total computation time (including family computation, pruning and partition generation). These numbers are obtained by fixing $\tau_l = 0.01/100$ for all G, while we set $\tau_h = 0.1/100$ for DBpedia, as we empirically tested that the resultant predicate set filters out both infrequent and heavy hitters. Likewise, we empirically set $\alpha_t = 0.05$ for both datasets, $\alpha_s = 0$ for WatDiv, and $\alpha_s = 0.01/100$ for DBpedia. Fig. 4.6 shows an ablation study in DBpedia to determine the number of generated families with different values of such parameters[15].

Table 4.2 also shows that $|F'_{core}|$, $|\mathcal{G}_{serv}|$, and the computation time are sub-linearly increasing with the graph sizes. In WatDiv, $F'_{core} = F'(G)$, whereas in DBpedia, the

[14] `https://psutil.readthedocs.io/en/latest/`

[15] We omit α_t as this study analyzes the size of families in the graph, while the extremely large families pruned by α_t tend to be generated when merging families.

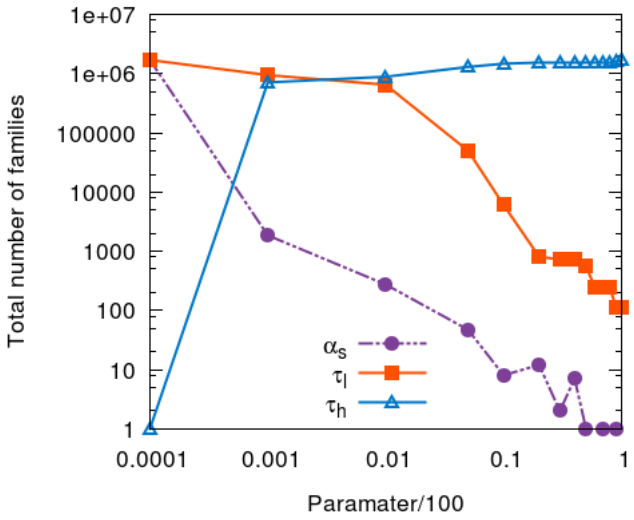


Figure 4.6: Ablation study in DBpedia to select the parameters in partitioning algorithm

initial number of P'_G-restricted[16] families $|F'(G)|$ is >600K: the family pruning strategy allows `smart-KG` to identify $|F'_{core}| = 35$ core families, which are merged into ∼30K materialised partitions.

4.5.2 Overall Query Performance

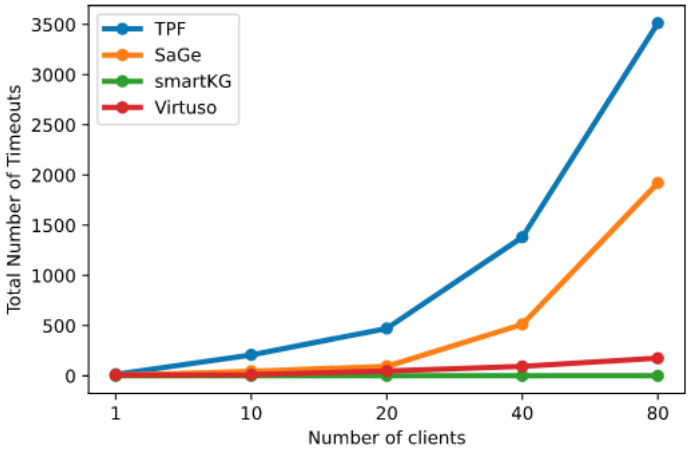
We report on performance for the WatDiv query workload, at increasing number of clients and dataset size. The performance of `smart-KG` always considers the family grouping/pruning strategies mentioned in Section 4.2.1.3; we also tested `smart-KG` without grouping/merging, which however did not scale, due to requiring shipping large numbers of small partitions with many redundant triples.

Increasing Number of Clients. In this part of the study, we focus on the graph WatDiv-100M as this is in line with the size of open KGs published in the LOD Cloud[17], with an average of 183M RDF triples. Fig. 4.7 shows the results of executing the WatDiv-100M workload on the query performance at a different number of concurrent clients (1, 10, 20, 40, and 80) in terms of (a) the number of timeouts, and (b) average workload execution time per client.

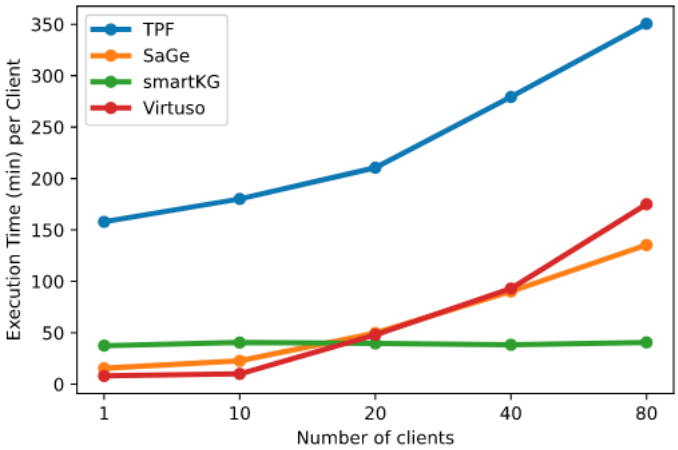
Fig. 4.7a shows that `smart-KG` produces no timeouts at such relative modest but state-of-the-art graph sizes. That is, even with 80 concurrent clients, our approach is able to successfully finish all queries in the workload for all concurrent clients. In contrast, TPF was not able to answer all queries within a 5 minutes timeout, even in the single client configuration. The percentage of timeouts escalates with increasing number of clients,

[16]The 218 restricted DBpedia predicates cover over 40% of the predicates occurring in highly-demanding BGPs (>1s of execution time) in the real-world LSQ query log [SAH+15].

[17]`https://lod-cloud.net`



(a) Number of timeouts



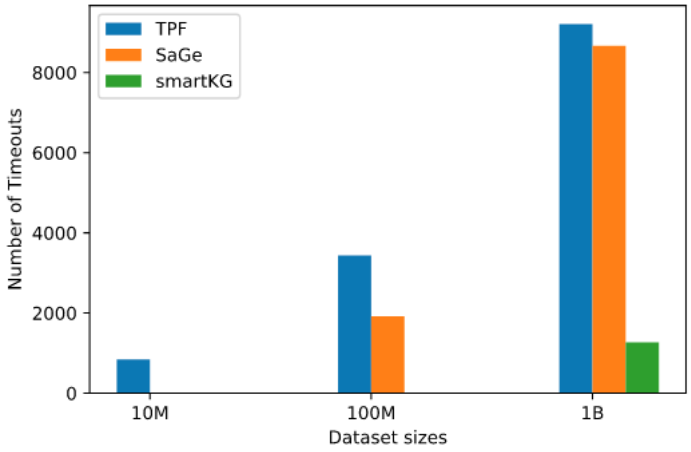
(b) Average execution time

Figure 4.7: Performance on the WatDiv-100M workload

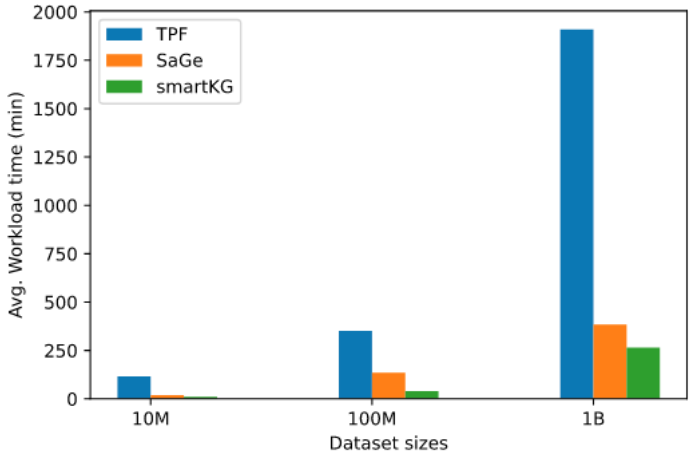
from 10% in 1-client workload to an average of 28% with 80 concurrent clients. These results confirm the scalability limitations of the system.

On WatDiv-100M, SaGe times out in fewer queries than TPF, but timeouts increase significantly with the number of clients, reaching a non-negligible 15% of the queries for 80 concurrent clients.

The average workload execution time per client, in Fig. 4.7b, shows superior performance and scalability of our approach, where performance remains constant irrespective of the number of clients, as `smart-KG` limits the server load and joins are mostly performed on clients over shipped KG partitions. For less than 20 concurrent clients, SaGe starts slightly ahead of `smart-KG`. From this point and on, SaGe suffers from excessive delays, and overall performance is degrading, e.g., `smart-KG` is up to 3.5 times faster with 80 clients. This is because SaGe executes SPARQL queries using a *round robin* policy to avoid the convoy effect but, with an increasing number of clients, the increased waiting



(a) Number of Timeouts



(b) Average Workload Execution Time

Figure 4.8: Performance on the workloads (80 clients) at increasing KG sizes

time and server usage lead to degrading average completion time for queries.

In turn, TPF is significantly worse – up to three times slower – than the other systems due to the enormous number of requests and the excessive data transfer. As we will show in the next section, traffic is substantially higher with larger datasets because clients need to ask for several server responses to evaluate a single query.

To complete the comparison, we also evaluate the performance of query shipping strategies using a Virtuoso SPARQL endpoint. As shown in Fig. 4.7, Virtuoso behaves very similar to SaGe with the difference that i) it shows no timeout and similar performance for 10 clients, but ii) its execution time is slightly degrading with 80 clients. Given these results and in line with previous studies [MSM19], in the following sections, we focus on comparing the performance of `smart-KG` with the shipping strategies of TPF and SaGe only.

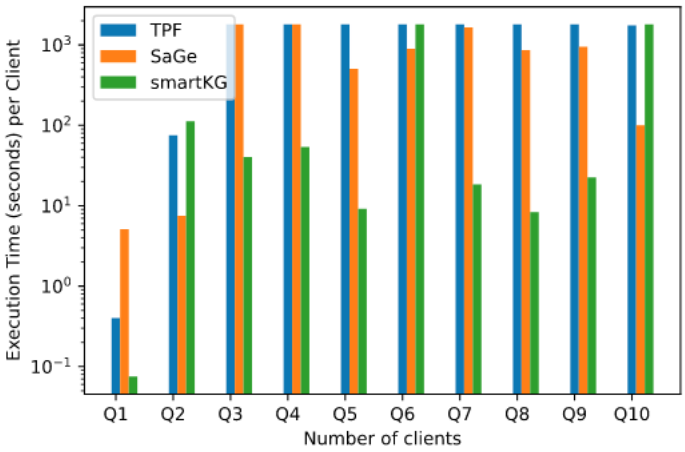


Figure 4.9: Average execution time (80 clients) with DBpedia high-demanding queries

Increasing KG Size. Fig. 4.8 shows the performance of the evaluated systems at increasing KG sizes, fixing the scenario to 80 concurrent clients. We execute the WatDiv workloads over 10M to 1000M triples, which constitutes, to the best of our knowledge, the largest experiment on client-side SPARQL query approaches to date.

Fig. 4.8a shows again the number of query timeouts. As expected, timeouts of TPF and SaGe significantly increase with the size of the graph. TPF is not able to scale for the WatDiv-1000M dataset, failing to answer 75% of the queries. Although SaGe is slightly better than TPF, it fails to answer 68% of the queries with 80 concurrent clients. In contrast, `smart-KG` reports the best results at scale, timing out only in the largest graph for 10% of the queries.

Fig. 4.8b presents the average workload execution time for all systems and different sizes, with 80 concurrent clients. As expected, average workload completion time increases with the larger KGs, while `smart-KG` remains the fastest in all scenarios. TPF has the longest execution time (and significantly longer in Watdiv-1000M) while SaGe is on average 1.5 times slower than `smart-KG` in Watdiv-1000M. Note also that average execution times include timeouts: since we have shown already that we do better in the number of timeouts, we may assume that full execution times with unlimited runtimes would be even more significantly in our favor.

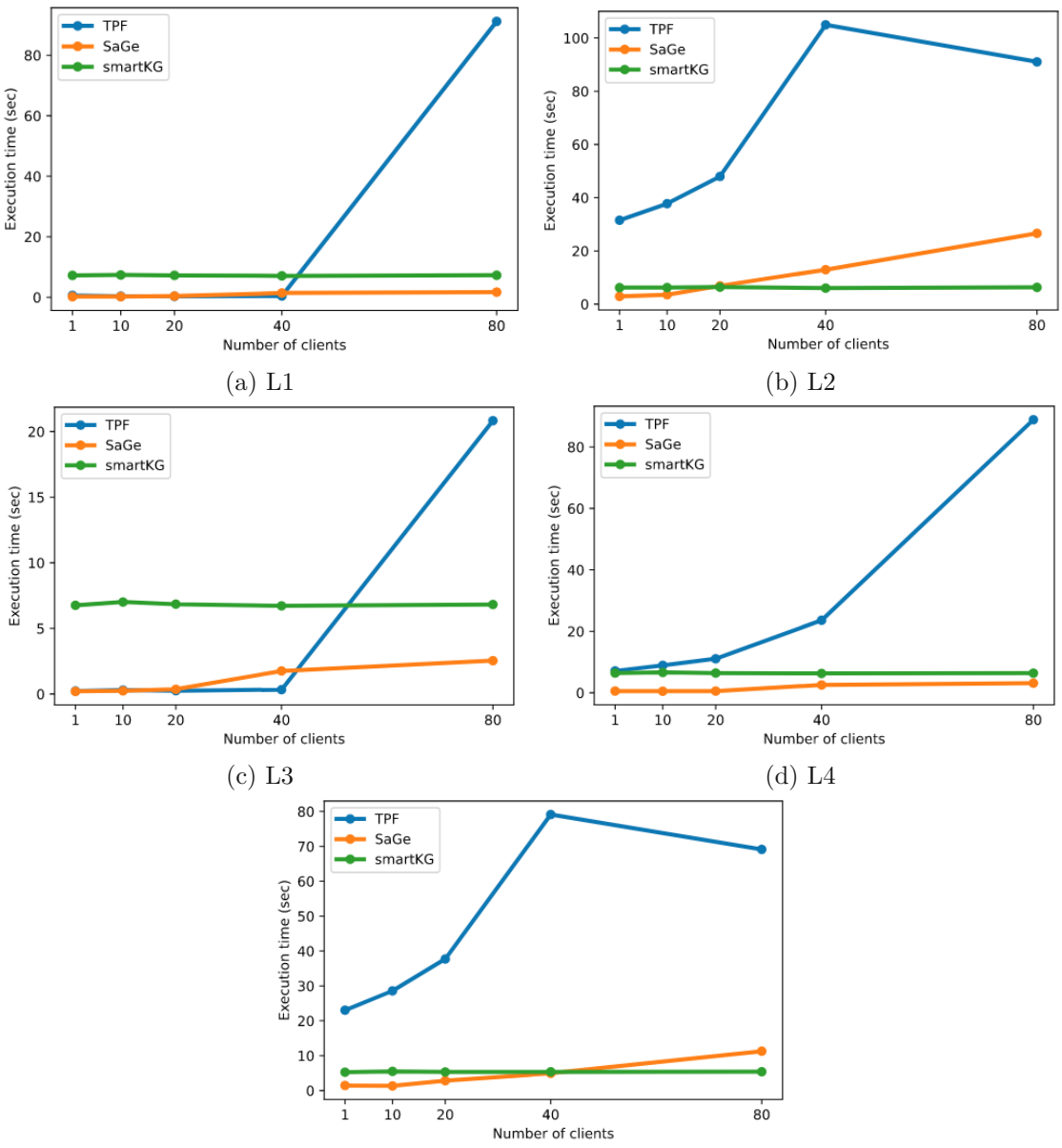
DBpedia Queries. We evaluate the performance on DBpedia to consider real-world data and high-demanding queries. Fig. 4.9 shows the performance results on 80 clients for 12 representative queries of the LSQ log, omitting Q11 and Q12 which time out (>30 minutes) in all systems. The results are in line with our previous analysis: TPF is the slowest (except Q1, Q2) and `smart-KG` is 2-3 levels of magnitude faster than SaGe in all queries except for Q2, Q6, and Q10. These are the cases where SmartKG depends heavily on TPF shipping, while SaGE can delegate to the SPARQL server.

4.5.3 Performance evaluation on different query shapes

While in the previous analysis, a workload consisted of queries with mixed characteristics, we have performed a separate performance analysis on two specific query categories predefined by the *WatDiv Basic Testing* [AHÖD14]: *linear (L)*, which represents simple path queries, and *Complex queries workload*, with more challenging queries including combinations of low-selective star and path queries. WatDiv provides L-query templates and we randomly generate 3 queries for each subtype (L1-L5) per client. The benchmark has only three C queries (not templates), hence, we extend it by selecting 50 complex queries (based on low selectivity patterns and high execution time) from the initial intensive workload, for each client.

Fig. 4.10 shows the performance in the simplest L-queries of the different systems on WatDiv-100M. Similar to our previous results, `smart-KG` reports a stable query execution time, which ranges between 5-7 seconds.

SaGe has the best performance in the L3 and L4 workloads, with average execution times of less than 2 seconds per query. However, the SaGe execution time is affected by the number of clients for L1, L2, and L5. SaGe provides better execution time than `smart-KG` with up to 20 concurrent clients, while `smart-KG` is more competitive for the larger number of clients. Finally, TPF is the slowest approach in L2, L4, and L5 queries, while it excels in L1 and L3 up to 40 clients. TPF could not scale to 80 clients.



(a) L1

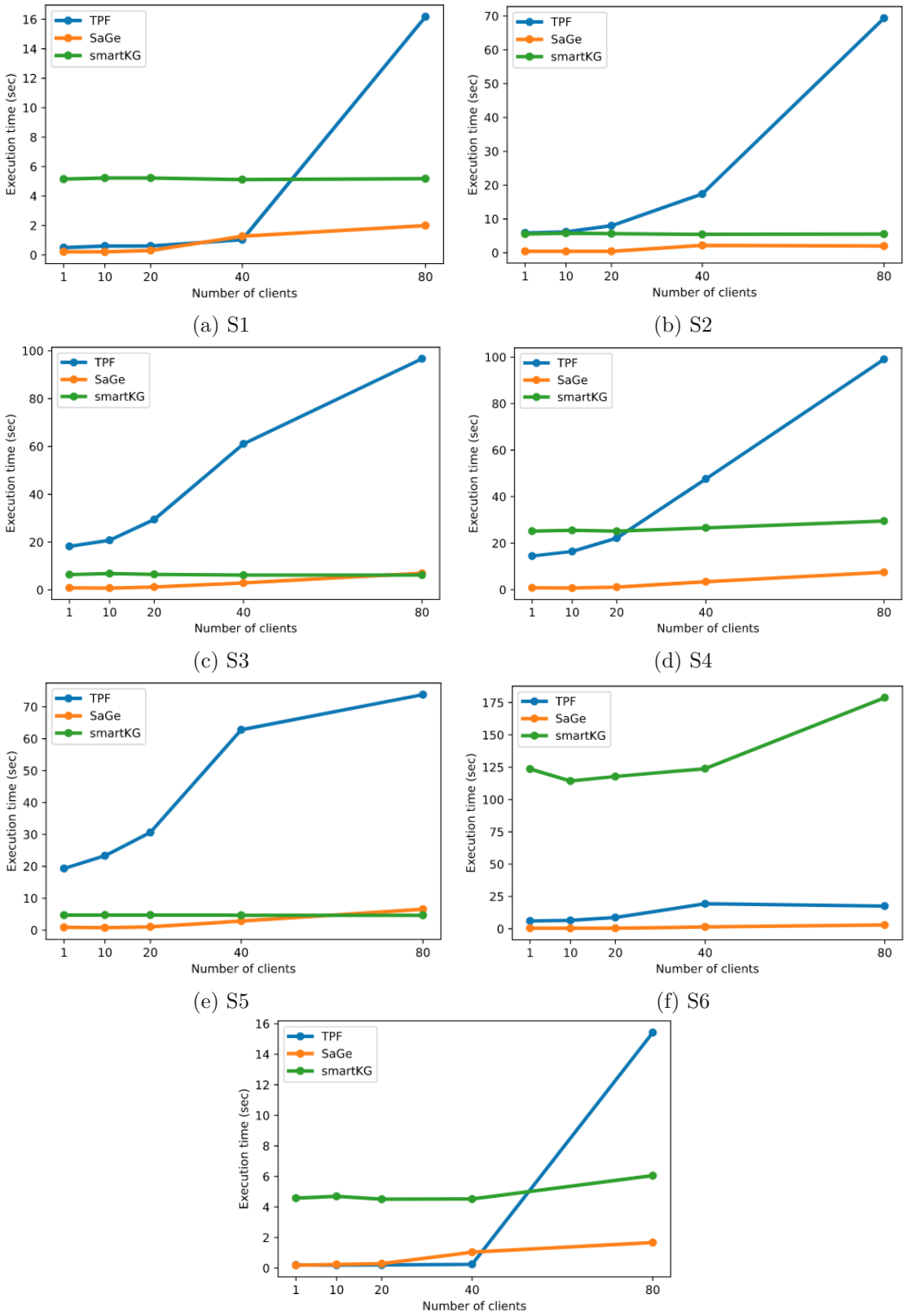
(b) L2

(c) L3

(d) L4

(e) L5

Figure 4.10: Avg. execution time per client on the standard WatDiv-100M for L queries



(a) S1

(b) S2

(c) S3

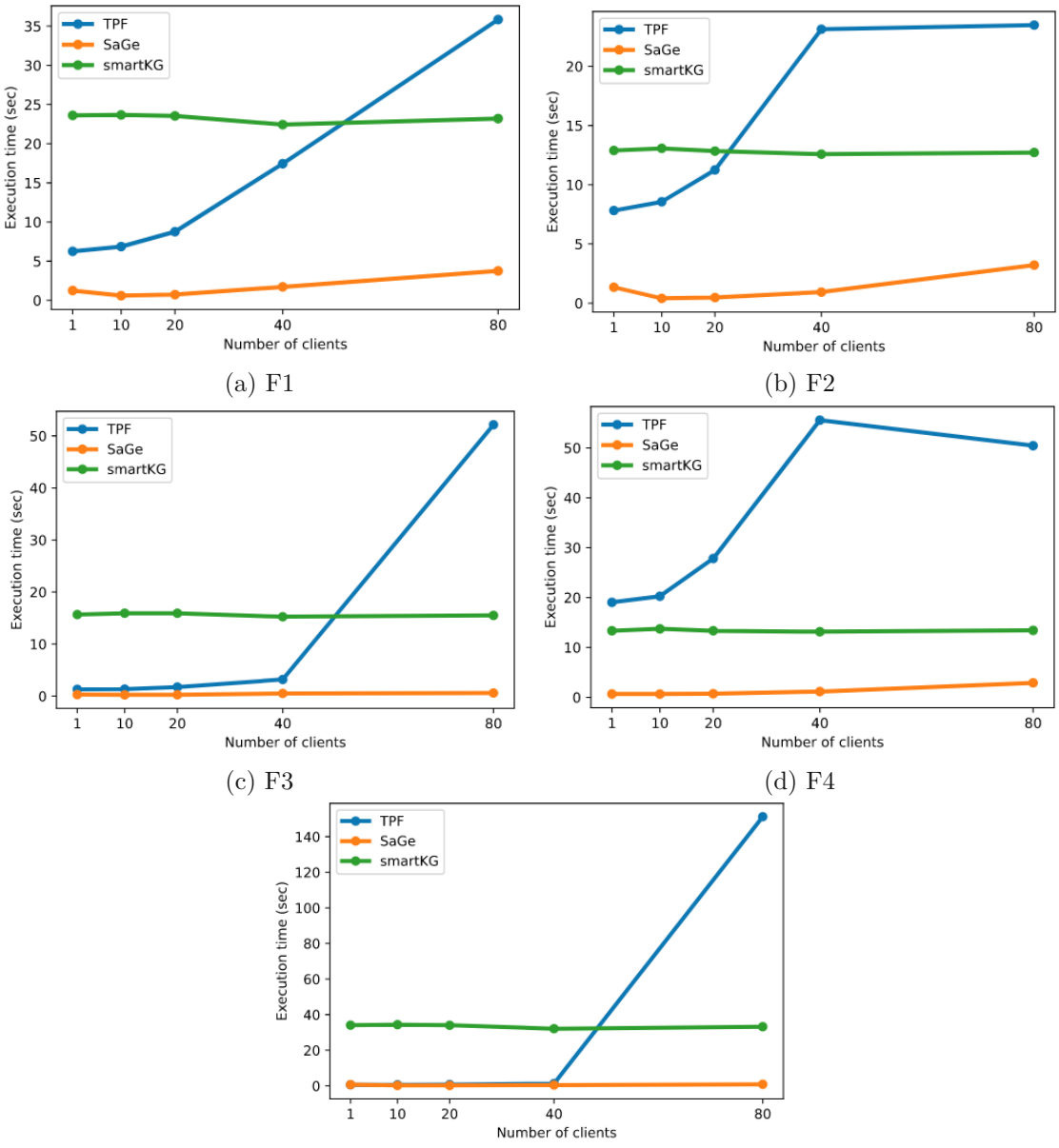
(d) S4

(e) S5

(f) S6

(g) S7

Figure 4.11: Avg. execution time per client on the standard WatDiv-100M for S queries



(a) F1

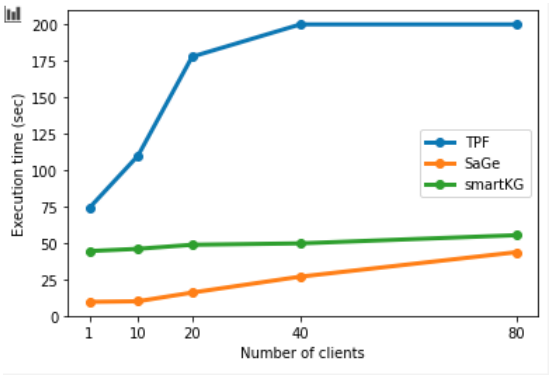
(b) F2

(c) F3

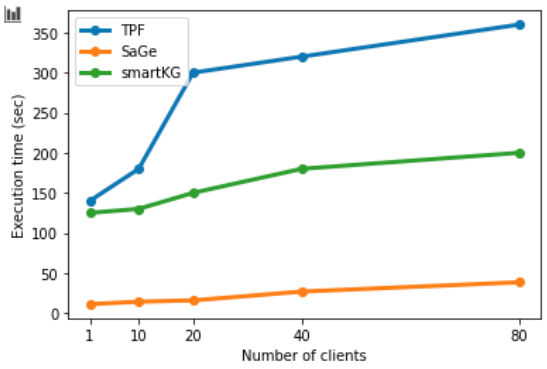
(d) F4

(e) F5

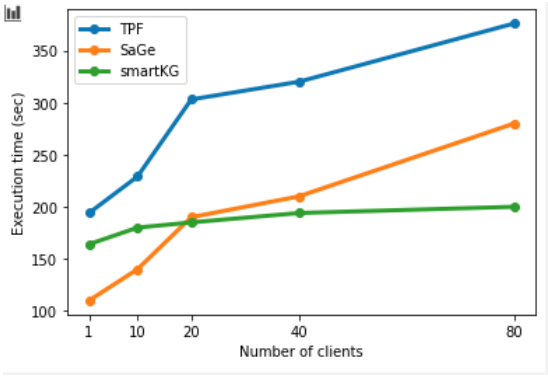
Figure 4.12: Avg. execution time per client on the standard WatDiv-100M for F queries



(a) C1



(b) C2



(c) C3

Figure 4.13: Avg. execution time per client on the standard WatDiv-100M for C queries

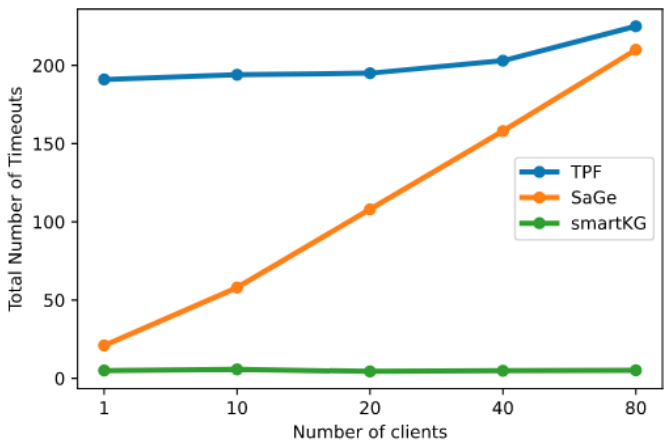


Figure 4.14: Complex Queries Workload

4.5.4 Resource Consumption

Server Network Load. Fig. 4.15a shows the average network traffic per client (in GB) on the intensive workload. We report the average per client, with 80 clients running in parallel on WatDiv-100M.

TPF incurs the highest communication costs due to the number of requests and the shipping of extensive intermediate results, which leads to poor query execution performance, as shown before.

In contrast, SaGe produces the least data transfer among all the systems, as it works as a SPARQL endpoint with a preemption model and no intermediate results are transferred. The only additional data transfer overheads in SaGe are the saved plans when queries are resumed, a relatively small cost depending on the number of calls required to finish each query. This factor depends on the complexity of the queries and the number of concurrent clients.

As expected, `smart-KG` requires more data transfer than SaGe, but up to 10× less data than TPF. Most of the data transferred is due to partition shipping (cf. Fig. 4.15a `smart-KG`-parts). Yet, the retrieved partitions can be reused for queries that require the same partitions. Caching the partitions will execute streak queries with minimal communication to the server. A streak is a concept defined in [BMT17] as sequence of queries that appear as subsequent modifications of a seed query.

Server CPU, RAM, and Disk Usage. Fig. 4.15b shows that TPF and SaGe extensively use the server CPU. In particular, SaGe and TPF respectively consume 80% and 60% of the CPU to execute 10 clients in parallel, and both rapidly increase to 100% for 40 and 80 clients. In practice, this reduces query throughput as most CPU time is allocated to query processing while new requests are queued. In contrast, `smart-KG` only uses 60% of the CPU to handle the workload on 80 parallel clients, which still gives room for serving additional clients in the current hardware configuration. `smart-KG` server consumes limited CPU thanks to its mixed triple pattern and partition shipping, which hardly requires server CPU usage.

Fig. 4.15c shows that TPF has the overall highest server memory usage, while SaGe's consumption remains constant and low, thanks to its preemption model. `smart-KG` uses the least memory consumption up to 20 clients and then slightly increases at 40 and 80 clients to 0.5 GB more than SaGe (due to TPF triple evaluation on the server).

We additionally compare to the resource consumption of Virtuoso, which shows relatively constant CPU (∼40%) and increasing RAM consumption, exceeding TPF for 80 clients. These values are in line with the significant timeouts reported previously (see Fig. 4.7).

Table 4.2 shows the raw data sizes (in N-Triples) of the graphs and storage requirements for the evaluated systems. As expected, TPF and SaGe require a single HDT file, which highly compacts storage needs. In contrast, the high number of HDT partitions managed by `smart-KG` results in additional costs in disk space, doubling the raw size of the WatDiv graphs (DBpedia uses less space given its more restricted pruning). Given that

Table 4.2: Comparison of storage requirements (in MB) for systems with HDT backend vs original graph size (raw)

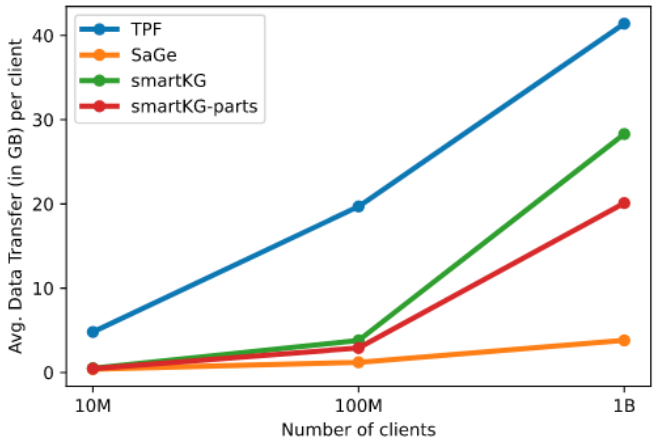
Dataset	Raw	SmartKG	TPF/SaGe
WatDiv-10M	1,471	2,783	112
WatDiv-100M	14,876	29,711	1,186
WatDiv-1000M	151,862	310,574	12,793
DBpedia	158,197	122,440	17,904

Virtuoso takes $\sim$ 3 times the space of TPF/SaGe.

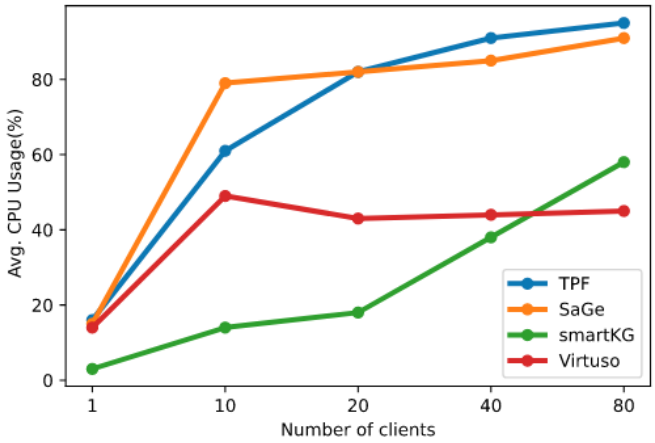
disk space is relatively affordable for servers, `smart-KG` provides a reasonable trade-off for faster and more balanced, SPARQL query execution.

Client CPU and RAM usage. As for client-side resources, as expected, Virtuoso excels with 80 clients in the WatDiv-100M workload (fully in the server, hence the clients run with 8% RAM size). SaGe also shows reasonable (average 15%) usage of the client CPUs, as it performs only two main operations on the client side: first, resuming query execution through received saved plans from the server; second, a subset of SPARQL operators such as filter, order by, and aggregations are offered. TPF performs joins of triple pattern results all locally on the client-side which is costly, leading to higher (55%) client average CPU usage. `smart-KG` finally also depends on the client to execute query stars as well as TPF Join processing, so that client average CPU usage is higher, with 70% yet visible for most of the current client systems.

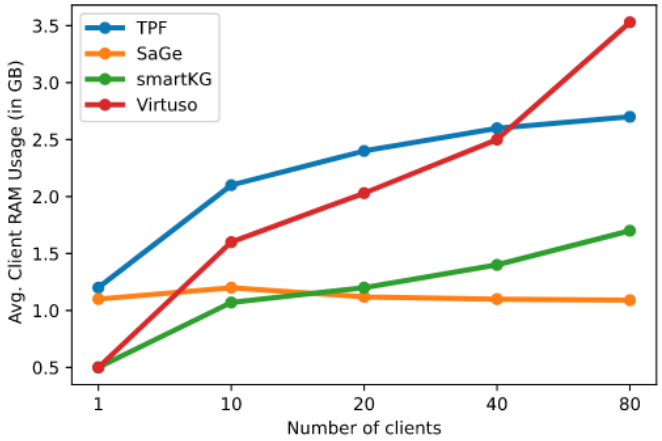
These percentages decrease with higher numbers of clients because network waiting time dominates in the case of TPF and `smart-KG` and the long waiting queues in the case of SaGe. Client memory consumption remains fairly constant and low for both SaGe and TPF. `smart-KG` consumes more client memory, however still reasonable. For instance, `smart-KG` utilizes up to 3 GB RAM.



(a) Network traffic per client (in GB) on the intensive workload at increasing KG sizes

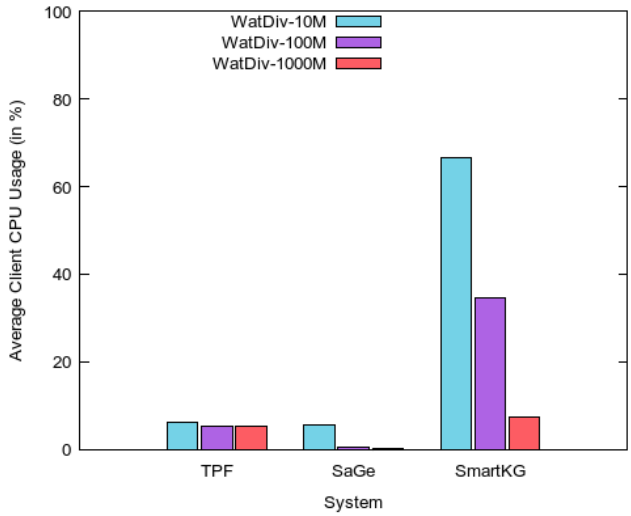


(b) Avg. Server CPU Usage (in %) at increasing number of clients (WatDiv-100M)

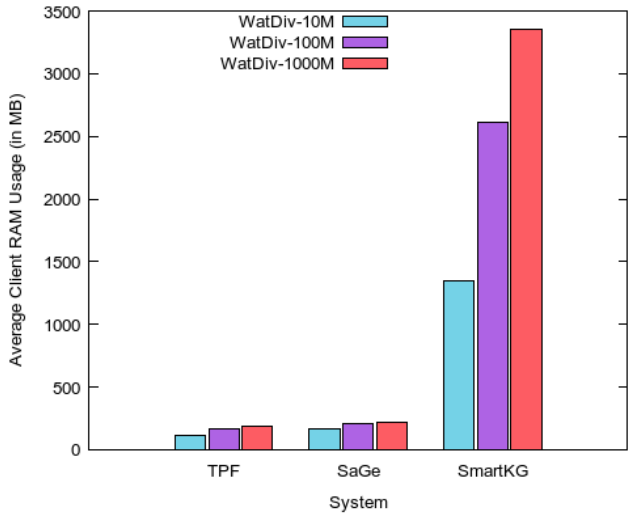


(c) Avg. Server RAM Usage (in GB) at increasing number of clients (WatDiv-100M)

Figure 4.15: Server-resources consumption on the intensive workload.



(a) Average Client CPU Usage (in %)



(b) Average Client RAM Usage (in MB)

Figure 4.16: Client CPU and RAM usage on the intensive workload at increasing sizes

4.6 Summary and Limitations

In this chapter, we have introduced `smart-KG`, a hybrid approach to efficiently query Knowledge Graphs (KGs) on the Web, balancing the load between servers and clients.

We combine the Triple Pattern Fragment (TPF) strategy with shipping compressed graph partitions that can be locally queried. On the server side, we consider a predicate family-based partition that regards the set of predicates describing each family, and we propose different heuristics to control the size of the partitions, as well as server operations to serve partition data and metadata. The served partitions are based on predicate families and different pruning parameters control the sizes and numbers of the partitions. The `smart-KG` client implements a query decomposer, planner, and executor tailored to trade off TPF and partition shipping. Our evaluation shows that `smart-KG` significantly outperforms the state of the art, especially with an increasing number of concurrent clients, and on challenging BGP queries. We also show that, at the cost of reasonable client resources, `smart-KG` improves server availability, consuming significantly less CPU and RAM than most of the evaluated systems, and reducing TPF's network traffic. However, we identify a number of shortcomings in our approach which hint at possible future directions:

- The creation of family-based partitions could potentially require the data publishers to have advanced skills in knowledge management in order to assign the best-fitting values to the pruning parameters used by the server (i.e., P'_{core}, τ_l, τ_h, α_s, and α_t). As a future work, an extensive analysis to the real-world open KGs is required to provide an automated parameter tuning according to the input KG [CW06].
- We argue (and have experimentally demonstrated) that our approach using family partitioning provides a reasonable trade-off of shipping sizes; still comparing to other partitioning strategies (e.g. predicate-wise, hash partitioning or a combination of multiple strategies) is on our agenda. In Chapter 6, we provide a generalization from the original LDF proposal to include (possible) implementation of Web querying interfaces based on existing partitioning techniques.
- The creation of family-based partitions is quite an expensive task. Possible improvements are as follows: (i) a multi-thread partition generation will potentially boost the generation performance, and (ii) Exploiting query logs of KGs served on the web to prioritize the generation of user-centered partitions [BMT17].

One of our core findings is that `smart-KG` only utilizes 60% of the server CPU to execute the workload on 80 parallel clients, which still gives room for further exploiting the server resources. In Chapter 5, we dynamically delegate the load between servers and clients for improved utilization of hardware resources.

CHAPTER 5

A Balanced Access to Web Knowledge Graphs

The availability of Web querying interfaces has been improved by `smart-KG`, which also demonstrates a competitive performance. However, current interfaces [VSH+16, AKMH20, MSM19], including `smart-KG`, utilize fixed load distribution strategies that may not be optimal in dynamic environments with varying workloads, network, and client and server capabilities.

Enabling Web querying over Knowledge Graphs is still impractical due to the imbalanced distribution of the query execution load between clients and servers. In this chapter, we present two possible strategies to dynamically distribute the query workload (i) an initial proposal that combines the strengths of various existing Linked Data Fragments (LDFs) based on a *predefined heuristic*, as well as (ii) `WiseKG` is a system that employs a cost model to dynamically delegate the load between servers and clients by combining client-side processing of shipped partitions with efficient server-side processing of star-shaped sub-queries, based on the current server workload and client capabilities.

By applying the cost model, servers can dynamically share the query processing tasks with the clients, making better use of server resources and retaining high performance even during high load. At the same time, they achieve significantly lower query processing times and by processing subqueries locally on the server, avoid unnecessary data shipping during periods with an overall low query processing load. `WiseKG` combines two approaches that have recently been proposed to optimize SPARQL query processing: Star Pattern Fragments (SPF), which exploits server-side evaluation of star-shaped subqueries, and `smart-KG`, our work from the previous chapter, which exploits client-side evaluation of star-shaped subqueries by retrieving compressed Knowledge Graph partitions from the server. By dynamically switching between these strategies based on the current server load and client capabilities.

Chapter Organization. The remainder of this chapter is organized as follows:

- In Section 5.1, we provide a motivating example.
- In Section 5.2, we give an overview of `WiseKG`, a novel system that dynamically shifts the query processing load between client and server, followed by a presentation of the server-side cost model. `WiseKG` employs the cost model to minimize the total time consumed by client-side and server-side components while considering the current load on the server and the client.
- In Section 5.3, we explain the collaboration between the `WiseKG` client and server to process SPARQL queries.
- In Section 5.4, we present an empirical evaluation of `WiseKG` using demanding query workloads on real-world KGs as well as synthetic KGs up to 1 billion triples shows that `WiseKG` significantly outperforms the state of the art.
- In Section 5.5, we summarize and outline the approach limitations.

5.1 Motivating Example

All thus far described KG APIs alone suffer from an imbalanced load on either the client-side (dumps, TPF, SKG) or server-side (SPARQL endpoints, `SaGe`, SPF). In this thesis, we, therefore, advocate that, based on decomposing BGPs into star-shaped subqueries and characteristics of these subqueries (e.g., selectivity and intermediate result cardinality estimation), we can optimally distribute the query processing load between client and server. Hence, given statistics as well as information about the current server workload and the client's capabilities, we can pick the best-suited KG API.

In particular, the factors that our cost model considers are server load, client computing resources, and the number/size of intermediate results to be transferred over the network (in combination with available bandwidth), since several sources [AKMH20, AFA+20, HA20, MVC+12, MKH19] identified these as important dimensions when accessing KGs.

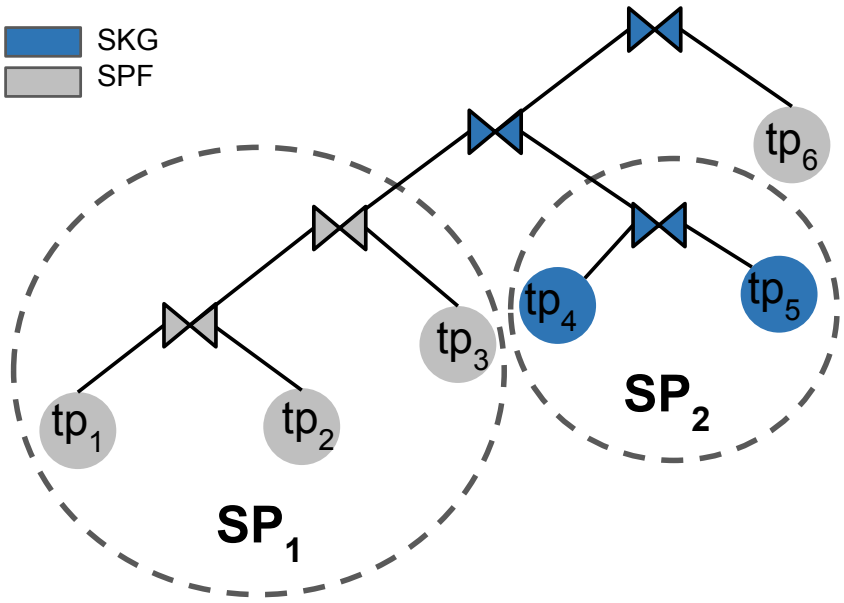
To elaborate, let us consider query Q given in Figure 5.1a. All triple patterns of Q have quite large cardinalities, meaning that both single pattern interfaces (TPF, brTPF) would need to send enormous numbers of requests to the server and ship large intermediate results to the client when processing the query.

For both star-based interfaces (SPF and SKG), the query would be decomposed into two stars and a single triple pattern: $sp_1 = \{tp1, tp2, tp3\}$, $sp_2 = \{tp4, tp5\}$, and $tp6$. sp_1 has 89,366 solution mappings, and sp_2 has 600,349 solution mappings. Both SKG and SPF would estimate the result sizes of star patterns and, in essence, order the query execution plan accordingly to (sp_1, sp_2, tp_6), i.e., starting with sp_1. SKG ships a partition containing 1,628,572 stars in total leading to excessive data transfer even though the partition is HDT-compressed. SPF, on the other hand, only ships the 86,366 stars

that actually match sp_1, resulting in less network overhead and faster query processing. However, in order to process the join between sp_1 and sp_2, SPF's client join processor would batch the 89,366 bindings into groups of 30 bindings each, sending one request per batch, amounting to 2,979 requests. This overhead could be conveniently mitigated by instead shipping the compressed partition for sp_2 and joining on the client: this example illustrates how a combination of SPF's server-side star evaluation with SKGs partition shipping could outperform either approach alone. Moreover, note that in case of a high server workload, the additional network overhead for transferring the partition for sp_1 might still be affordable, compared to server-side SPF processing of sp_1 using the overloaded server.

```
select * where {
  ?album dbo:artist ?artist . # tp1: 146,716 matches (sp1)
  ?album rdf:type dbo:Album . # tp2: 147,917 matches (sp1)
  ?album dbo:releaseDate ?date . # tp3: 212,290 matches (sp1)
  ?artist dbo:genre ?genre . # tp4: 576,000 matches (sp2)
  ?artist foaf:name ?name . # tp5: 4,146,579 matches (sp2)
  ?song dbo:writer ?artist . # tp6: 200,969 matches
}
```

(a) Show artists' albums, genres, and the songs they have written



(b) Query execution plan for $(sp_1^{SPF}, sp_2^{SKG}, tp_6^{SPF})$

Figure 5.1: Example of processing a SPARQL query with `WiseKG`

5.2 WiseKG

In the spirit of the example presented in Section 5.1, WiseKG enables to leverage (i) the characteristics of the star-shaped subqueries as well as (ii) information on the currently available client and server resources, to estimate the cost of processing each star-shaped subquery on the client (using SKG) or on the server (using SPF), – choosing the most efficient execution strategy dynamically.

5.2.1 Overview

WiseKG employs a dynamic cost model to determine an *annotated query plan*: in order to denote query execution plans with particular interfaces to be used per subquery, we will use superscripts SPF and SKG, i.e., for our example the annotated plan $\Pi = (sp_1^{SPF}, sp_2^{SKG}, tp_6^{SPF})$. In the case of the example in Figure 5.1b this would mean that sp_1 is evaluated via SPF on the server, sp_2 is executed using SKG on the client, and the resulting bindings from joining both are given as input Ω to a call of tp_6 executed again using SPF on the server[1].

Upon receiving a BGP P from the client, the WiseKG server will decompose it into star-shaped subqueries, and use its cost-model to create an annotated query plan Π that is returned to the client, along with a timestamp τ denoting plan expiry. The client then, in the order specified by the server, executes Π using the APIs specified in the plan annotations. In case the execution is not completed by τ, the client needs to request a new annotated plan, which may look different – as mentioned before and illustrated in the example, the choice of API per subquery taken by the server may depend on its current load, as discussed in the following.

Formally, the WiseKG server API offers the following interface calls to access KG G:

- an SPF LDF API control $SPF(P, \Omega)$ returning $s_{SPF}(G, P, \Omega)$,
- an SKG LDF API control $SKG(P, \Omega)$ returning $s_{SKG}(G, P, \emptyset)$[2],
- an execution plan interface $Plan(P)$ returning a $pair(\Pi_P, \tau)$.

We will use the notation $c(P, \Omega)$ to denote that a (star-shaped) sub-pattern P is executed by a control $c \in \{SPF, SKG\}$ – in the spirit of LDF, we expect also other (hypermedia) controls to be callable in addition to SPF and SKG in the future. Further, we assume that the call to $c(P, \Omega)$ on the client side is converted to a set of bindings through a function $eval_c(P, \Omega) = \Omega \bowtie [[P]]_G$. Note that, depending on whether the underlying selector function of $c(P, \Omega)$ is already accepting bindings, directly returning $\Omega \bowtie [[P]]_G$ (such as for SPF) or only returning a graph of which $[[P]]_G$ can be computed and then

[1] Note that for triple patterns, SPF is equivalent to brTPF so we can use the SPF interface also for single triple patterns.

[2] Note that SKG does not allow to ship bindings, cf. Section 4.2.2.

joined with Ω on the client (such as for SKG), $eval_c$ incurs more or less work on the client side.

$Plan(P)$ maps a BPG P to an annotated plan Π_P along with the expiry timestamp $\tau = \tau_{\mathcal{C}} + \iota$, where $\tau_{\mathcal{C}}$ corresponds to the current time, and ι is a fixed time quantum per query[3]. Π_P is constructed from $\mathcal{S}(P)$ by (i) identifying the best join amongst stars based on cardinality estimations and (ii) determining, based on factors such as the current load on the server and the estimated network/processing cost, the best interface (SPF or SKG) per subquery. Before we explain (server and client) query processing in more detail (cf. Section 5.3), we first present the server cost model, which is used to make this latter choice.

5.2.2 Server-Side Cost Model

In this section, we present WiseKG's server cost model used to determine the choice between client-side evaluation using SKG or server-side evaluation using SPF. The cost model is inspired by the classic R^* optimizer [ML86] from the field of distributed databases [ML86, ZMG+20]. In the R^* model, the total time is the sum of four-time components (CPU processing, messaging, data transfer, and I/O) that can be estimated for a query Q as:

$$cost(Q) = processing + Messaging + data\ transfer + I/O$$

Following the R* model, we consider, in our client-server architecture, the following components to approximate the total time consumed by the client and server to process a star subquery: the estimated number of CPU instructions ($\#CPU$), the estimated number of I/O operations ($\#IO$), as well as two communication cost components – the estimated number of requests ($\#M$) and estimated number of transferred bytes ($\#BYT$) over the network per query. WiseKG's cost model for a given star subquery is then defined as

$$\begin{aligned} cost(sp) = & \underbrace{W_{CPU} \times (\#CPU)}_{\text{Processing}} + \underbrace{W_{MSG} \times (\#M)}_{\text{Messaging}} \\ & + \underbrace{W_{BYT} \times (\#BYT)}_{\text{Data transfer}} + \underbrace{W_{IO} \times (\#IO)}_{\text{I/O}} \end{aligned} \tag{5.1}$$

where the weights W_{CPU}, W_{MSG}, W_{BYT}, and W_{IO} help estimate the time required by the client and server hardware configuration to perform a CPU instruction, the time required to send an (HTTP) request message from a client to a server over the network, the time required to transfer one byte from a server to a client over the network, as well as the time required for a disk I/O operation. It is important to note that WiseKG's server optimizer is tailored to embed dynamic factors to reflect the current server load. These weights are estimated as follows:

[3] Somewhat similar to/inspired by SAGE's[MSM19] query suspension timeouts.

W_{CPU}: We estimate time per CPU instruction as the inverse of the CPU's IPS (Instructions per second) rate, damped by the current CPU load in percent[4]:

$$W_{CPU} = \frac{1}{IPS \times (100\% - CPU_{usage})}$$

W_{MSG}: The average time to transmit an HTTP request from a client to the server. In our experiments and network setup, similar to SaGe's experiments [MSM19], we assume a constant value of $W_{MSG} = 50ms$ for all clients. In a real-world scenario, we would measure this delay based on an initial HTTP request per client.

W_{BYT}: We estimate W_{BYT} by the conservative minimum between the available server bandwidth bw_{serv} (which we estimate as the difference between the bandwidth of the server network card reduced by the average data transfer over the network in the last 1 minute, again checking every second) and the client bandwidth bw_{client}, which we estimate as $20Mb/sec$ in our setup, similar to [AFA+20]. This way, W_{BYT} takes into account the current network usage of concurrent clients. In our experiments,

$$W_{BYT} = \frac{1}{Min(bw_{client}, bw_{serv})}$$

W_{IO}: We measure I/O in terms of loading chunks of 1MB from disk, i.e., we estimate W_{IO} as the time required to read 1MB to the memory. In WiseKG, the I/O times differ per chosen API: for SPF, a single HDT file of the entire graph G is used and mapped into memory while auxiliary bitmap indexes remain in memory to help localize potential mapping solutions (using approx. 3% of the entire HDT file altogether [FMG+13]). Thus, the I/O time accounts for transferring non-cached blocks that might contain the mapping solutions to memory. In SKG, the I/O time is due to the server reading HDT *partitions* from disk in order to ship those to the client; on the client side, we assume processing continues in memory, thus not involving further I/O operations.

We note that our experiments have shown that in fact, I/O is a negligible factor in our setup; for both SPF and SKG (we perform a respective experiment with a stress-testing workload described in Section 5.4.1), we verified that the amount and difference in I/O times in both approaches was dwarfed by the communication costs. Therefore, we leave out this factor in our cost estimation model ($W_{IO} = 0$).

The final time cost estimates of client-side SKG evaluation based on shipped partitions vs. server-side SPF evaluation of star patterns are given in Definition 5.1 and Definition 5.2. For a query BGP P, these costs are estimated for each star pattern $sp \in \mathcal{S}(P)$.

[4]We estimate this current CPU load as the average percentage of CPU_{usage} in the previous minute (checking every 1sec). Note that for our experiments we only compute this CPU usage on the server side, i.e. for $W_{CPU_{serv}}$, whereas for $W_{CPU_{client}}$ we assume $CPU_{usage} = 0$, i.e., full availability of client resources.

Definition 5.1 (Cost of SKG Star Pattern Evaluation)**.** *Given a star pattern* $sp \in \mathcal{S}(P)$ *and a plan* Π_P*, as well as the set of families* $F_{sp} = \{f \in F(G) \mid f \supseteq pred(sp)\}$ *relevant for* sp *in* G*, the cost in time of evaluating* sp *on using SKG is estimated as follows:*

$$cost_{SKG}(sp, \Pi) = W_{CPU_{client}} \times \underbrace{card(sp, \Pi) \times i_t}_{\#CPU} + W_{MSG} \times \underbrace{|F_{sp}|}_{\#M} + W_{BYT} \times \underbrace{\left(\sum_{f \in F_{sp}} size(f) \right)}_{\#BYT} + W_{I/O} \times \underbrace{\left(\sum_{f \in F_{sp}} size(f) \right)}_{\#IO}$$

Definition 5.2 (Cost of SPF Star Pattern Evaluation)**.** *Given* sp*,* Π_P*, and* F_{sp}*, the cost in time of evaluating* sp *using SPF is estimated as follows:*

$$cost_{SPF}(sp, \Pi) = W_{CPU_{serv}} \times \underbrace{card(sp, \Pi) \times i_t}_{\#CPU} + W_{MSG} \times \underbrace{\frac{card(sp, \Pi)}{\Phi(n)}}_{\#M} + W_{BYT} \times \underbrace{card(sp, \Pi) \times b_t}_{\#BYT} + W_{IO} \times \underbrace{size(G)}_{\#IO}$$

Definitions 5.1 and 5.2 use the following functions and variables:

- $card(sp, \Pi)$ returns an estimated result cardinality for evaluating star pattern sp using an estimation of the number of bindings for previously evaluated star patterns in Π. This estimate (based on statistics about the sizes of subgraphs per characteristic set) is described in [NM11].
- $size(\cdot)$ is either the size of an HDT file (plus index) for a partition corresponding to a family $f \in F(G)$ or, for $size(G)$ the size of the HDT file for the entire graph G[5].
- i_t is the number of CPU instructions needed to process each triple in the result set. In general, we rely on HDT algorithmic costs which are sub-linear and close to constant for most operations [FMG+13]; we only measured one millisecond (or at most a few milliseconds) in our experiments. We therefore set this factor to $i_t = 1$. Different IPS rates in the server and client are considered in the different weights: $W_{CPU_{serv}}$ and $W_{CPU_{client}}$.

[5] Note that SPF relies on a single HDT for G whereas SKG only transfers the HDT files corresponding to F_{sp}.

- b_t is the average number of bytes per triple in the result; we estimate this factor by averaging the size of the triples in each family partition.

5.3 Query Processing

In this section, we detail how the `WiseKG` server and client work together to process SPARQL queries. In particular, we describe how the query processing is performed on the server side and on the client side.

5.3.1 Server-Side Query Processing

Since the server-side processing of star-shaped subqueries in SPF and SKG APIs running on the server are explained in detail in [AFA+20] and [AKMH20], we mainly focus on the creation of the annotated execution plan in this section: when the `WiseKG` server receives a $Plan(P)$ request for a BGP P, it creates a query execution plan specific to P, which it returns along with the expiry timestamp τ to the client for execution; the resp. algorithm to compute $Plan(P)$ is shown in Alg. 5.1.

The first step is to decompose the query into star-shaped subqueries (line 2). To create the execution plan, we find the star-subquery with the lowest cardinality estimation (line 5-8) and add it to the plan; when we find a query with an empty result (e.g. in case no matching family partition exists [AFA+20]), we can stop since the final result will then also be empty. The star pattern with the lowest cardinality estimation is selected first (line 9), thus overall in the final plan, patterns are ordered by estimated cardinality.

Then, the estimated costs for SPF and SKG are compared in Line 10; depending on the cost models from Section 5.2.2, each subquery is annotated with the resp. control for evaluating the star pattern on the server, i.e., SPF (line 11) or the client SKG (line 13). Here, the *append* function just appends the annotated star pattern to the end of the plan. When there are no more subqueries left in the star decomposition, the algorithm returns the plan (line 16) after computing the expiry timestamp (line 15).

For the query Q shown in Figure 5.1a, this algorithm could compute the execution plan in the join order visualized in Figure 4.4c (unless the server load is too high, in which case SP_1 could also potentially be suggested to be executed using SKG).

Finally, as a side note, we note that based on the fact that not all family partitions in SKG are necessarily materialized on the server – SKG does not materialize HDT files over a certain partition cardinality threshold (for details, cf. [AFA+20]); in such cases, the concrete implementation of Alg. 5.1 defaults to SPF, i.e., server-side evaluation of the resp. star pattern, independent of the cost.

5.3.2 Client-Side Query Processing

Processing queries on a `WiseKG` client relies on an approach similar to the one presented in [AKMH20], which we adapt herein to accommodate for client-side processing of HDT

Algorithm 5.1: Create an annotated query execution plan

Input: $P = \{tp_1, tp_2, \ldots, tp_n\}$ // a BGP
Output: (Π_p, τ) // an annotated plan and its expiry time

```
1  Function Plan(P)
2      S ← S(P)
3      Π_p ← ()
4      while S ≠ ∅ do
5          for sp ∈ S do
6              cnt_sp ← card(sp, Π_p)
7              if cnt_sp = 0 then
8                  return ()
9              end
10         end
11         sp_i ← sp where sp ∈ S and cnt_sp ≤ cnt_sp' for all sp' ∈ S
12         if cost_SPF(sp_i, Π_P) ≤ cost_SKG(sp_i, Π_P) then
13             Π_P ← append(Π_P, (sp_i^SPF))
14         end
15         else
16             Π_P ← append(Π_P, (sp_i^SKG))
17         end
18         S ← S \ {sp_i}
19     end
20     τ ← τ_C + ι
21     return (Π_P, τ)
22 end
```

shipped family partitions. In the following, we describe the basic ingredients that the client needs to process full SPARQL queries: `WiseKG` is able to process full SPARQL queries including operators such as `UNION` and `OPTIONAL`, `FILTER`, etc.,[6] which are all evaluated on the client-side. Herein, we only focus on the BGP evaluation part.

The general approach for processing BGPs P is as follows:

1. Retrieve the query execution plan and time quantum for P from the server by calling $Plan(P) = (\Pi_P, \tau)$.

2. For each star pattern $sp^c \in \Pi_P$ with control $c \in \{SPF, SKG\}$ in Π_P and solution mappings from previously evaluated operators Ω, iteratively do the following:

   a) If $\tau < \tau_C$, i.e., the plan has expired, the client requests a new execution plan/expiry based on the remainder of P that has not yet been processed.

[6] with the exception of GRAPH query patterns, since HDT does not support named graphs.

b) Otherwise we call the interface $c(sp, \Omega)$ and convert it to a set of bindings using $eval_c(sp, \Omega)$, which as mentioned above, in the case of $c = SKG$ involved client-side evaluation of the star-shaped pattern on the shipped HDT, whereas SPF directly returns the result bindings.

The exact algorithm implementing these steps in a recursive manner is shown in Alg. 5.2.

Algorithm 5.2: Processing a Query Execution Plan

```
   Input: Π = (sp_1^{c_1}, ..., sp_n^{c_n})  // an execution plan;
          τ                                  // expiry timestamp;
          Ω'                                 // a set of bindings
   Output: Ω                                 //set of solution bindings
 1 Function evalPlan(Π, τ, Ω)
 2     if τ < τ_C then
 3         (Π, τ) ← Plan(BGP(Π))
 4     end
 5     if Π = sp^c then
 6         Ω ← eval_c(sp, Ω')
 7     end
 8     else
 9         Ω ← evalPlan((sp_1^{c_1}, ..., sp_{n-1}^{c_{n-1}}), τ, Ω')
10         Ω ← evalPlan(sp_n^{c_n}, τ, Ω)
11     end
12     return Ω
13 end
```

Line 2 checks whether the plan has not yet expired; in that case, the algorithm calls $Plan(\Pi)$ to reevaluate the plan on the server (line 3)[7]. The way this is currently done can be understood as follows: assuming the originally requested plan is $(sp_1^{c_1} \dots sp_i^{c_i} \dots sp_n^{c_n})$ and the client reaches τ at step i. Then the client will restart calling $Plan(\{sp_i, \dots sp_n\})$ receiving a new plan $\Pi_{\{sp_i, \dots sp_n\}}$ upon which it continues; obviously this could change the interface choices per star for the remaining plan, based on the current server load situation. Continuing on Alg. 5.2, in case the plan is associated with a single star pattern sp (line 4), we call the control $c \in \{SPF, SKG\}$ to retrieve the output plan and obtain the output solution mappings (line 5). Otherwise, the algorithm will make a recursive call for the left subtree (line 8) the resulting bindings of which are handed over to the call of the right subtree (line 9).

[7] Here, $BGP(\Pi)$ denotes the corresponding (non-annotated) BGP for plan Π.

5.4 Experimental Evaluation

In this section, we compare the performance of `WiseKG` with the state-of-the-art SPARQL query processing interfaces.

5.4.1 Experimental Setup

In this section, we describe the experimental setup, including the systems we compare against, datasets, queries, and hardware and software configurations.

Implementation details. We implemented both `WiseKG` client and server in Java[8] extending the TPF implementations[9] so that we ensure comparability and compatibility with the spectrum of Linked Data Fragment (LDF) approaches including TPF, SPF, and smart-KG. The `WiseKG` server relies on SPF star pattern fragments for server-side processing of star subqueries. Furthermore, the `WiseKG` server adopts the family generator component from smart-KG [AFA+20] to generate, manage, and store the HDT files of the family-based partitions. In our server-side cost model, we depend on a cross-platform operating system and hardware information library for Java[10] to retrieve system information about clients and the server resources usage including network and CPU usage. The `WiseKG` client implements a pipeline of nested iterators similar to brTPF and SPF client implementations.

Configuration. To assess the performance of our system under different loads, we perform experiments over eight configurations with 2^i clients ($0 \leq i \leq 7$) issuing queries concurrently for each configuration (up to 128 concurrent clients). Each concurrent client executes one query at a time, i.e., at most 128 queries are executed at the same time.

Datasets. We use three different sizes of the Waterloo SPARQL Diversity Benchmark (WatDiv) [AHÖD14] to test the scalability of our approach: 10M, 100M, and 1B triples. In addition to these, we also use the real-world dataset DBpedia [LIJ+15] (v.2015A). The characteristics of the evaluated RDF graphs are described in Table 4.1.

Queries. We consider three different query workloads for the WatDiv datasets: (i) a *basic testing* workload named `watdiv-btt` that consists of queries obtained from WatDiv basic testing templates[11]. Each client has a set of 20 queries including star queries (S), linear queries (L), snowflake queries (F), and complex queries (C); and (ii) a diverse *stress testing* workload named `watdiv-sts` that consists of queries obtained from the WatDiv stress-testing suite[AHÖD14]. Each client has a set of 154 non-overlapping queries. In addition to these workloads, we randomly selected 16 queries from a real-world LSQ query log [SAH+15]; plus, we included 12 queries used to evaluate smart-KG [AFA+20].

Compared Systems. To test the effectiveness of dynamically shifting star-subquery processing between client-side and server-side based on the status of server-side resources

[8] `https://github.com/WiseKG/WiseKG-Java`
[9] `https://github.com/LinkedDataFragments/Server.java`
[10] `https://github.com/oshi/oshi`
[11] `https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml`

disregarding the cost model defined in Section 5.2.2, we implemented a version of WiseKG named $\texttt{WiseKG}_{heuristic}$ that relies on more straightforward heuristics. Initially, $\texttt{WiseKG}_{heuristic}$ executes all star subqueries on the server side up to a predefined CPU usage threshold σ. When the threshold is reached, $\texttt{WiseKG}_{heuristic}$ produces an execution plan exclusively based on shipping family partitions. In addition, we evaluate $\texttt{WiseKG}_{cost}$, our main contribution, which is a version of WiseKG that relies on the cost model described in Section 5.2.2. Note that we use the recommended versions of both server and client for all the evaluated systems including Star Pattern Fragment (SPF) [AKMH20], smart-KG [AFA+20], SaGe [MSM19], and Triple Pattern Fragments (TPF) [VSH+16].

Hardware configuration. We ran all 128 clients concurrently on a virtual machine with 128 2.5GHz vCPU cores, 64KB L1 cache, 512KB L2 cache, 8192KB L3 cache, and 2TB main memory. To ensure an even distribution of the resources between the clients, we limited each client (for all approaches) to run with a single vCPU core and 15GB of main memory. WiseKG and all the compared system servers were run on the same server with 32 3GHz vCPU cores, 64KB L1 cache, 4096KB L2 cache, 16384KB L3 cache, and 128GB main memory. Clients and servers are located on the same 1 GBit network. In order to emulate a more realistic bandwidth scenario, we limited the network speed of each client to 20 MBit/sec.

Evaluation metrics.

- **Timeouts**: number of queries that exceed the timeout.
- **Workload Completion Time**: the total time required by a client to complete a workload.
- **Query Execution Time**: the average execution time for takes to complete a query.
- **Server CPU load**: the average percentage of server CPU usage during the execution of a query workload.
- **Number of Requests made to the Server**: the number of requests a client sends to the server.
- **Number of Transferred Bytes**: the number of bytes transferred between server and client, i.e., the sum of both directions.

Software configuration. Following the experiments performed in [MSM19, AFA+20, AKMH20], we used a timeout of 300 seconds, i.e., 5 minutes, for all approaches. That is, after 5 minutes we suspend the query execution. The page size $\Phi(n)$ for TPF, SPF, and WiseKG was set to $n = 100$ (as in [AKMH20, VSH+16]) and the maximum number of bindings attached to a request for SPF and WiseKG was set to $|\Omega| = 30$ as it was in [AKMH20]. In order to assess our approach against the others using as similar as

possible configurations, we set the time quantum ι to the same value as the overall timeout for all systems, i.e., 5 minutes.[12]

5.4.2 Experimental Results

All results, incl. additional experiments, details on the implementation and configurations used in the experiments (datasets and queries) are available online[13].

System Performance Evaluation. In this part of the evaluation, we focus on analyzing the behavior of the compared systems in the scenario of increasing KG size with the highest number of concurrent clients (128 clients) using the `watdiv-sts` workload. As shown in Figure 5.2, $\texttt{WiseKG}_{heuristic}$, the vanilla version of `WiseKG`, performs significantly better than the state-of-the-art systems in terms of performance and scalability, not to mention $\texttt{WiseKG}_{cost}$ (just `WiseKG` hereafter) has even surpassed $\texttt{WiseKG}_{heuristic}$.

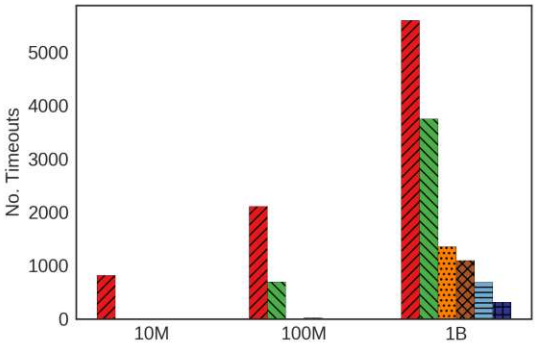
Figure 5.2a shows that `WiseKG` produces no timeouts over the `watdiv10M` and `watdiv100M` datasets for 128 concurrent clients. Moreover, even in the case of `watdiv1B`, `WiseKG` only incurs 2% timeouts of the total workload queries. In contrast, none of the compared systems was able to process all queries with a 5-minute timeout, except SPF and `SaGe` on the `watdiv10M` dataset. When queries are executed over the `watdiv1B` dataset, the percentages of timeouts reach 13% and 21% for smart-KG and SPF, respectively. For `SaGe` and TPF, the percentages of timeouts increase up to 55%. These results confirm the superior scalability of `WiseKG` compared to state-of-the-art systems. These experiments show that even for a high number of clients, `WiseKG` is able to handle large scale KGs.

Figure 5.2b shows the average workload completion time including queries that timed out. `WiseKG` is up to 4 times faster than SPF and smart-KG, and up to an order of magnitude faster than `SaGe` and TPF over `watdiv1B` with a load of 128 concurrent clients. In addition, Figure 5.2b also shows that SPF and smart-KG have comparable average workload time. smart-KG performs slightly better for `watdiv100M` and `watdiv1B` datasets. This is not surprising since they similarly rely on star decomposition; SPF executes the star subqueries on the server side while smart-KG ships the relevant partitions for the subqueries and executes them on the client.

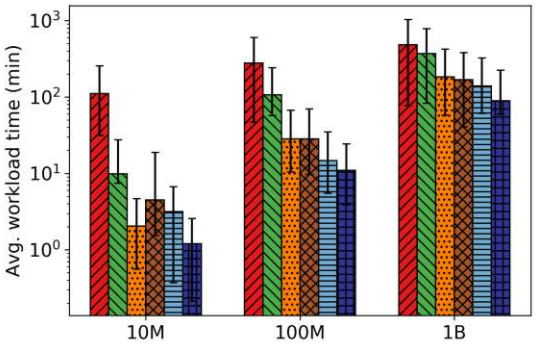
Compared to SPF and smart-KG, `WiseKG` provides a significant performance improvement as a result of the proposed cost model that optimizes query processing by leveraging the subqueries' cardinality estimation as well as available client and server resources to determine an efficient execution plan.

[12] In our current setup and evaluation covering widely used benchmarks in the area, the expiry timestamp was hardly reached. While we already significantly outperform all state-of-existing approaches, we still deem the addition of a plan expiry needed both conceptually (as the system resources change dynamically over time and our model needs to consider the current "promises" it made to clients) and useful for future workloads on larger knowledge graph.

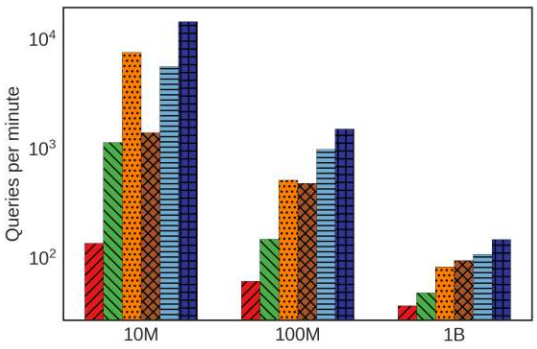
[13] `https://github.com/WiseKG/WiseKG-Experiments`



(a) Number of Timeouts



(b) Avg. Workload Time



(c) Queries per minute

Figure 5.2: Number of timeouts, average workload time, and throughput for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` on `watdiv-sts`

To provide a comprehensive evaluation, we also include TPF and `SaGe` in our experiments. As shown in Figure 5.2, our experiments confirm a previous study [AFA+20] that `SaGe` performs far better than TPF for small datasets. However, when dataset size increases and the number of concurrent clients is high, the difference between TPF and `SaGe` becomes less visible. Note that we did not include a SPARQL endpoint (e.g Virtuoso) in

our experiments, since several previous studies [VSH+16, MSM19, AFA+20, AKMH20] have already shown that SPARQL endpoints are not able to scale well with an increasing number of clients.

We compare the performance of WiseKG to state-of-the-art interfaces considering real-world queries on DBpedia. Figure 5.3 presents the execution times of these 28 queries for all systems. The results confirm that WiseKG significantly outperforms the compared systems for real-world queries. Figure 5.3 shows that TPF is the slowest or the second to slowest in all queries. On the one hand, smart-KG suffers from excessive delays in queries that require non-materialized partitions such as Q2, Q4, Q8, Q12, Q15, Q19, Q21, and Q25 since, in this case, smart-KG depends on TPF in addition to queries with high selectivity such as Q6, Q16, Q20, and Q26 as it is more resource-efficient to process on the server-side. On the other hand, SPF has a robust performance in most of the queries due to its efficient server-side star pattern execution, except the queries with low selectivity such as Q24 and Q28 due to the excessive transfer of intermediate results.

Moreover, SaGe has worse performance than WiseKG for the less selective queries with large intermediate results, such as Q7 and Q28, due to these queries putting more load on the server and incurring more requests to the server. The queries where SaGe has slightly better performance than WiseKG, such as Q2 and Q4, are generally queries where the overhead of computing the execution plan for WiseKG is a considerable part of the overall execution time (i.e, very simple queries). Finally, $\texttt{WiseKG}_{heuristic}$ is faster than WiseKG for the queries with an execution time of less than 0.1 seconds. This is because WiseKG has the overhead of computing the best query plan.

Performance evaluation on different query shapes. In this part of the evaluation, we analyze the effect of the query shapes on the performance of the systems. We use query workloads consisting of 4 shapes including linear (L), star (S), snowflake (F), and complex (C) shapes. These queries are part of the watdiv-btt workload and executed against watdiv100M. The queries of each workload were executed in a different (random) per client, averaging in the results the overall execution times per workload across all clients. Figure 5.4 shows the average query execution time for each shape.

In compliance with the system performance analysis, WiseKG outperforms all state-of-the-art systems for all different query shapes. For the L-workload, all systems have a similarly efficient performance since this workload includes the simplest queries with a small diameter. As shown in Fig. 5.4b, SPF provides excellent performance for S-workload – as expected since it is optimized for star queries with high selectivity. On the other hand, smart-KG performs worse than SPF since it sends an entire partition with unnecessary intermediate results for such queries. In general, SaGe has an outstanding performance for all query shapes, especially for the F-workload as shown in Fig. 5.4c. This is due to the fact that the watdiv-btt workload includes only 20 queries per client (i.e low query arrival rate) and we use a medium-size watdiv-100M dataset for this experiment.

Fig. 5.4d shows that the behavior of the compared systems dramatically changes for the C-workload. For instance, WiseKG significantly outperforms state-of-the-art interfaces,

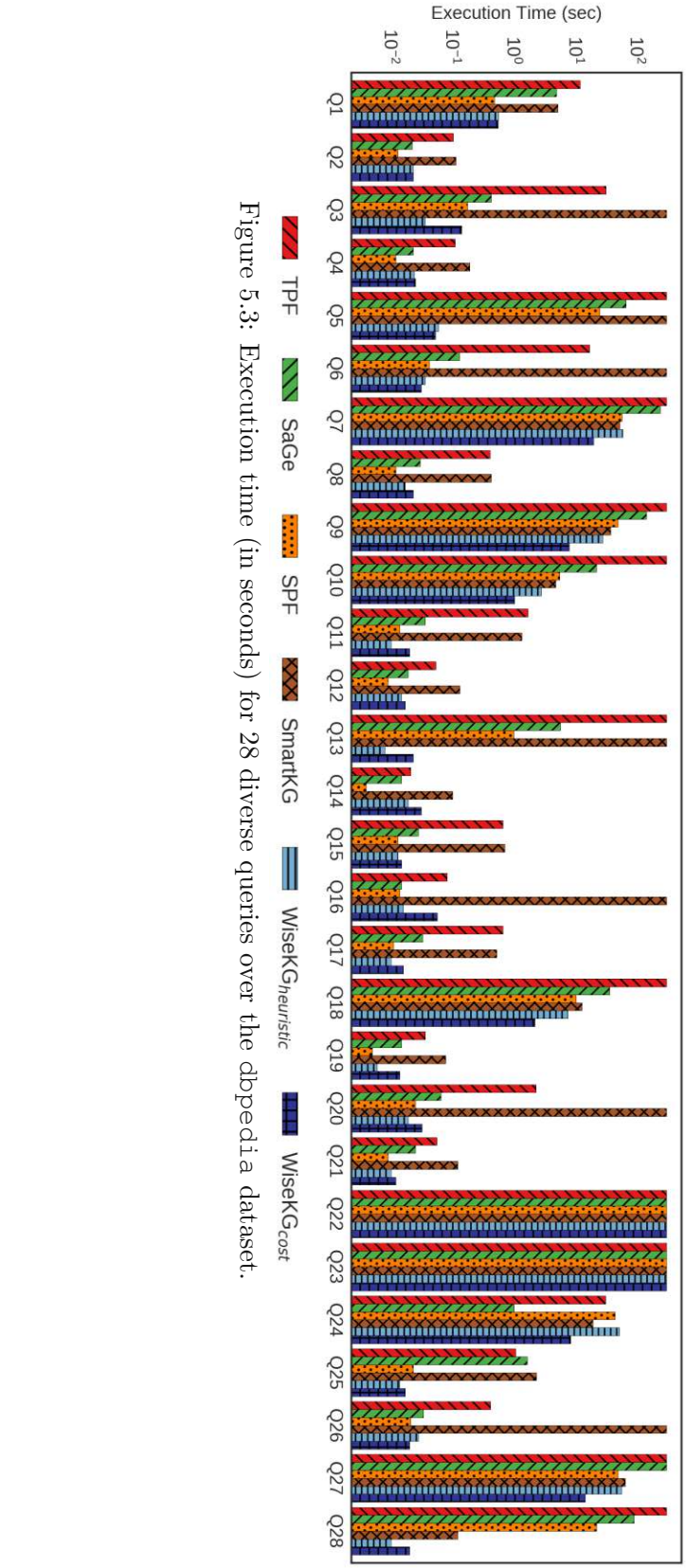
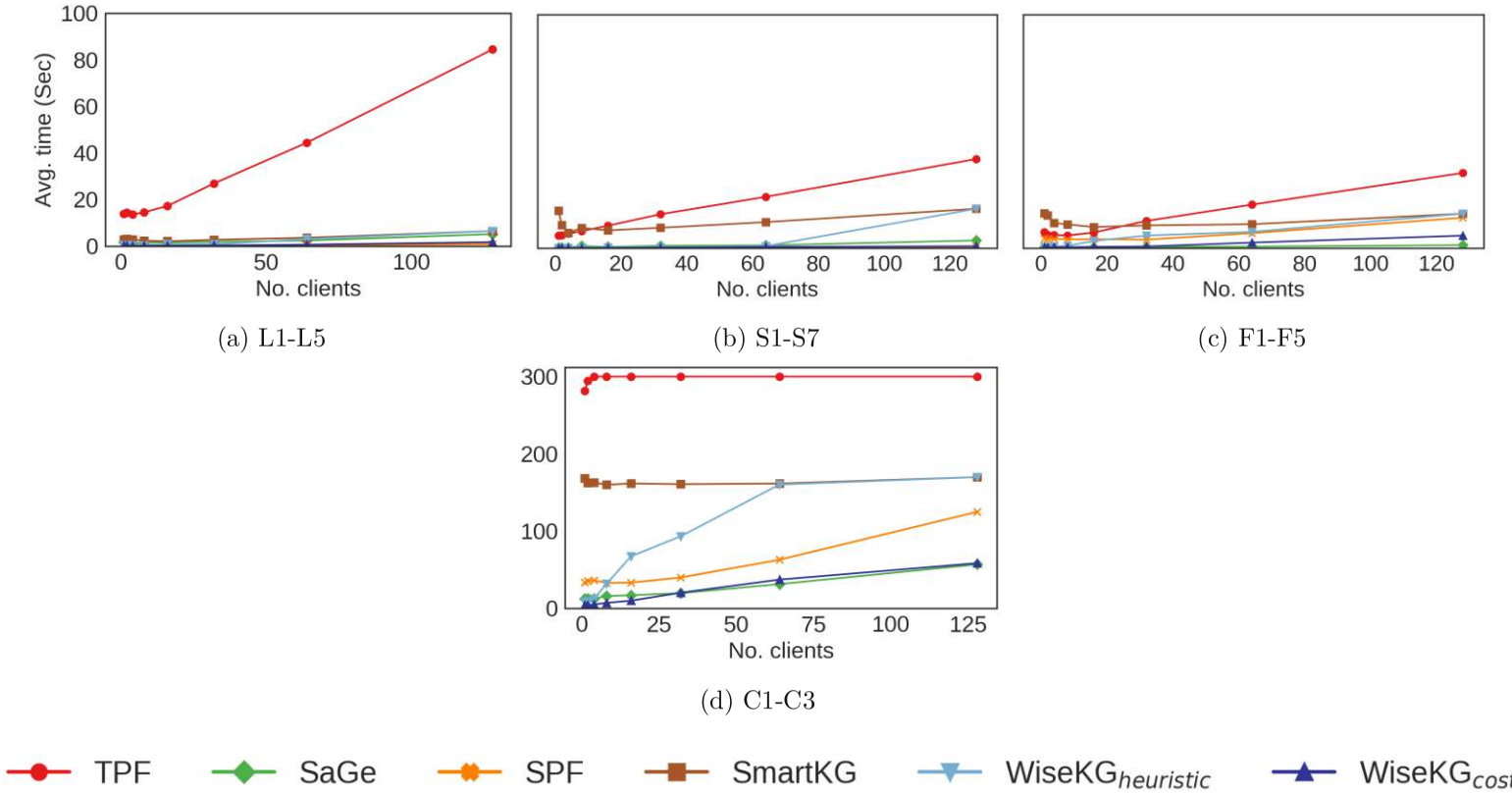


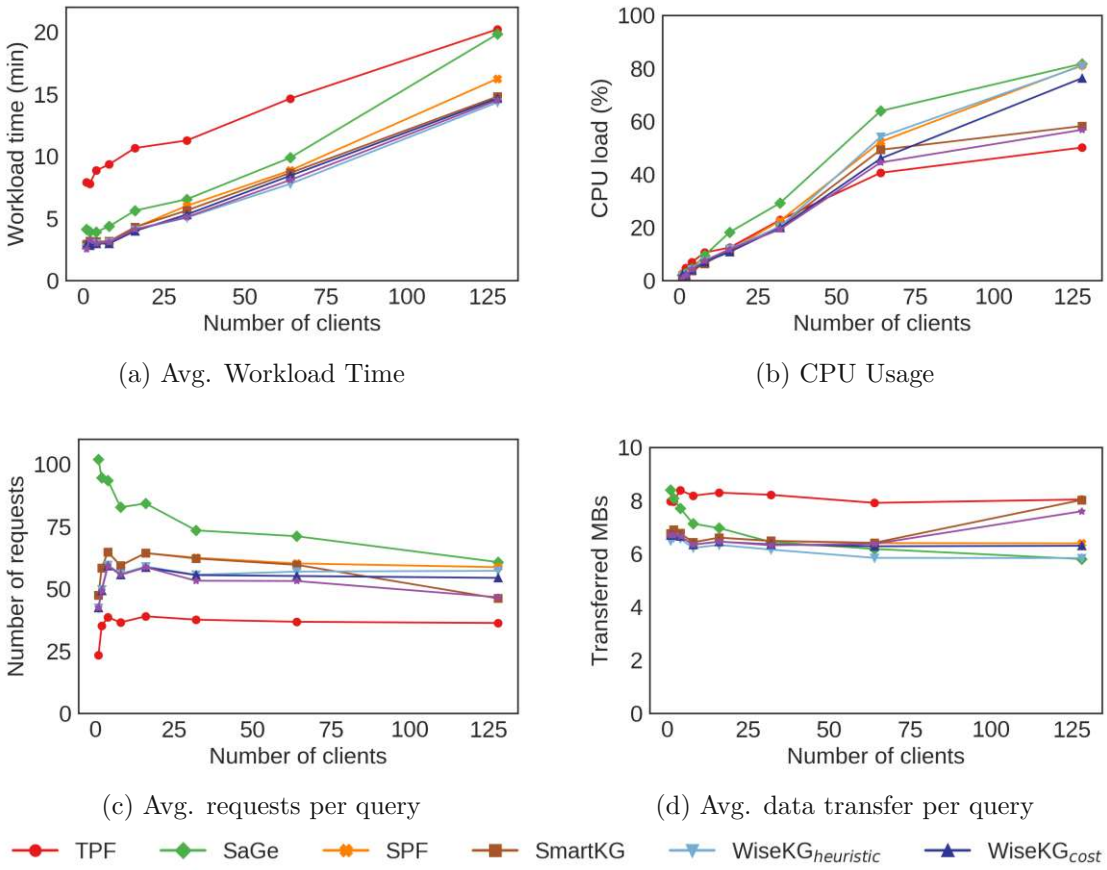
Figure 5.3: Execution time (in seconds) for 28 diverse queries over the dbpedia dataset.



(a) L1-L5 (b) S1-S7 (c) F1-F5

(d) C1-C3

Figure 5.4: Avg. execution time per client over `watdiv100M` for the `watdiv-btt` workload.



(a) Avg. Workload Time

(b) CPU Usage

(c) Avg. requests per query

(d) Avg. data transfer per query

Figure 5.5: Impact of the cost model components on the performance and resources consumption over `watdiv100M`

even `SaGe` in the single client configuration. `SaGe` starts ahead of smart-KG up to 16 clients, then smart-KG performs better with higher numbers of concurrent clients. SPF suffers excessive delays in C1 since the query includes 3 stars that have intermediate results with high cardinalities. For query C2, `SaGe` outperforms all the compared systems. In contrast, smart-KG and TPF are significantly worse (both time out) than SPF due to SPF's better handling of triple patterns with large cardinalities by shipping bindings along with star-shaped subquery requests. Interestingly, although `WiseKG`$_{heuristic}$ times out in C2, `WiseKG` was able to efficiently perform the query with a slightly higher average time compared to `SaGe`. This is due to the accurate estimations of the cost model. Finally, for C3, though SPF and smart-KG are optimized for star queries, e.g., C3 is a single unbounded star, `WiseKG` is up to three times faster with 128 clients.

Impact of cost model components. We performed an experiment with several different configurations of the cost model over `watdiv100M` on the `watdiv-sts` workload in order to evaluate the impact of the cost model components on `WiseKG` query performance and resource consumption. To measure the impact of the cost model components, we

configured three different versions of WiseKG including the data transfer component only ($Cost_D$), data transfer and messaging components ($Cost_{MD}$), and finally, a version with processing, messaging, and data transfer components ($Cost_{PMD}$). For this experiment, we used $\texttt{WiseKG}_{heuristic}$ as a baseline. Figure 5.5a shows that for the configuration with 128 clients $Cost_{PMD}$ improves the average workload completion time (14 min) compared to $Cost_D$ and $Cost_{MD}$ ($19min$ and $16min$, respectively). In addition, Figures 5.5b and 5.5c show that $Cost_{PMD}$ requires on average less CPU usage and number of requests than $Cost_D$ and $Cost_{MD}$. This is due to the fact that the $Cost_{PMD}$ configuration includes the processing component which significantly contributes to lowering the CPU load on the server. Although $Cost_D$ has the lowest transferred data compared to the rest of the configurations, $Cost_D$ is the slowest configuration. The reason for this behavior is that it does not take into account the HTTP request latency, which is an important factor to determine the incurred latency especially, in subqueries that require high numbers of result pages. It is important to note that all the configurations remain faster than $\texttt{WiseKG}_{heuristic}$, and since $\texttt{WiseKG}_{heuristic}$ is faster than all the state-of-the-art systems (Figure 5.2), so are all the configurations.

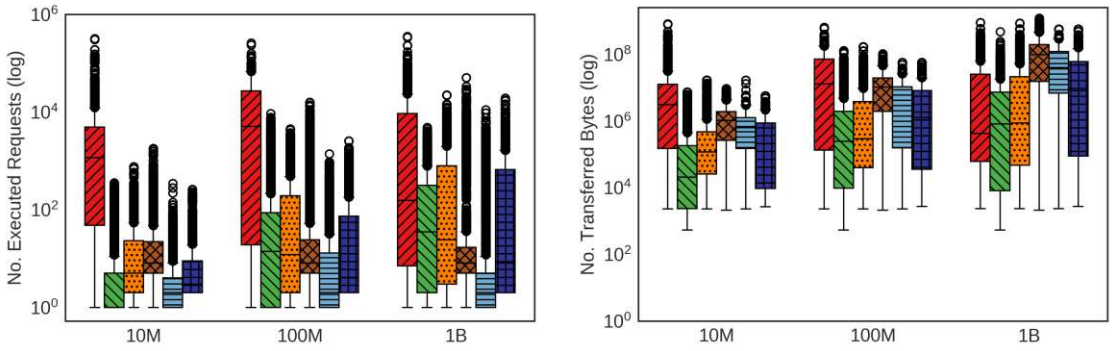
Moreover, to evaluate the impact of using *characteristic set* [NM11] as a cardinality estimation method on the cost model components, we replaced the cardinality estimation function in the WiseKG configurations described earlier with the true cardinality, creating the configurations $Exact_D$, $Exact_{DC}$, and $Exact_{PMD}$, respectively. Figures 5.5b, 5.5c, and 5.5d show that $Exact_D$ and $Exact_{DC}$ provide faster performance and better resource utilization compared to their peers with cardinality estimation $Cost_D$ and $Cost_{PMD}$. Figure 5.5a shows that the configurations with the true cardinality have a comparable workload execution time ($\approx 14min$). This performance is similar to the performance of $Cost_{PMD}$ even though $Exact_{PMD}$ has a lower resource consumption.

Finally, our experimental results show that relying on characteristic sets as a cardinality estimation method provides comparable performance to the configurations with the true cardinality – demonstrating a very subtle impact of the cardinality miss-estimates on the overall performance of WiseKG. We plan to investigate diverse cardinality estimators as future work in order to explore the impact of different cardinality estimation techniques on WiseKG query execution time [LGM+15, PKB+20].

Resource consumption. In this part of the evaluation, we focus on the server resource usage including network and CPU consumption.

We report two main metrics to demonstrate the network traffic: the number of requests sent to the server (NRS) and the number of transferred bytes between the client and server (NTB). Figures 5.6a and 5.6b show the distribution of the number of requests to the server per query as well as the distribution of the number of transferred bytes per query, with 128 concurrent clients on increasing KG sizes (watdiv10M, watdiv100M, and watdiv1B) for the watdiv-sts workload. As expected, TPF incurs the highest number of requests and transferred data, leading to a substantial increase in network load. Even though smart-KG relies on TPF to execute singular triple patterns and star

patterns with no materialized partition, smart-KG significantly reduces the number of requests compared to TPF since it only sends a single request per star pattern.



(a) Number of requests to the server for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` (log).

(b) Number of transferred bytes for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` (log).

Figure 5.6: Number of requests to the server and number of transferred bytes for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B`, and CPU load for increasing numbers of clients over `watdiv1B` on the `watdiv-sts` workload

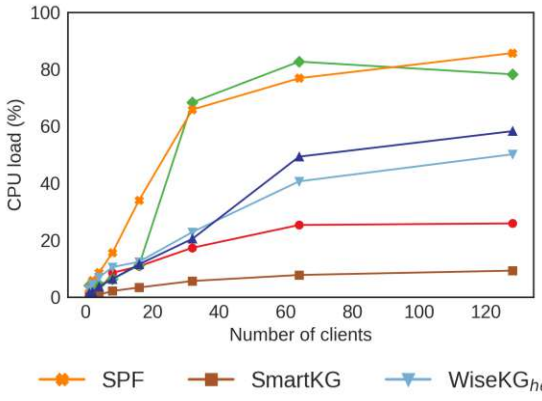


Figure 5.7: Avg. Server CPU Usage (in %) for increasing numbers of clients over `watdiv1B`.

Figure 5.6a also demonstrates that `WiseKG` requires the lowest average number of requests among all systems due to three main reasons: first, `WiseKG` potentially reduces the number of requests required based on the communication component in the cost model which can be observed in the difference between the number of requests $Cost_D$ and $Cost_{DC}$ as shown in Figure 5.5c; second, `WiseKG`, in contrast to smart-KG, ships bindings along with the triple pattern requests (as presented in brTPF [HA16] that

requires fewer requests than TPF); third, WiseKG has an advantage over SPF to require less requests in case of star patterns with low selectivity. Figure 5.6b shows that SaGe incurs the least data transfer among all compared systems since SaGe is essentially a SPARQL endpoint with a preemption model that only transfers the final results. As expected, WiseKG incurs less data transfer than TPF, smart-KG, and SPF. To be precise, WiseKG transfers on average 5.5MB per query while SPF and smart-KG transfer 7MB and 13MB over watdiv100M dataset. WiseKG demands on average less intermediate results than SPF and smart-KG thanks to the cardinality estimation aware cost model.

Figure 5.7 presents the average server CPU usage per system when the watdiv-sts workload is executed over the watdiv100M dataset. SPF and SaGe consume more CPU on the server side. This is expected since SPF processes star pattern queries on the server side and SaGe utilizes a SPARQL endpoint that does all the work on the server side. As one can see from Figure 5.7, the CPU usage of these two interfaces approach the CPU processing capabilities when the concurrent number of clients is set to 128. In contrast, CPU consumption of smart-KG and TPF remain almost constant and quite low; under 20% and 30%, respectively. This low consumption is inline with restricted capabilities of these servers: partition shipping in case of smart-KG and triple pattern lookup in case of TPF.

Figure 5.7 shows that WiseKG's CPU usage is almost in the middle between SPF and smart-KG, where it gradually increases up to 60% in the case of 128 concurrent clients, which enables WiseKG to serve more queries given the current server capabilities (Figure 5.2a).

5.5 Summary and Limitations

We introduced WiseKG, a querying interface to efficiently access Web Knowledge Graphs. We propose an efficient query processing approach under high query loads by balancing the SPARQL query execution load between servers and clients. To this end, we have combined two Linked Data Fragments APIs (SPF and smart-KG) that enable server-side and client-side processing of star-shaped sub-patterns. Our dynamic cost model picks the best-suited API per sub-query based on the current server load, client capabilities, estimation of necessary data transfer between client and server (for intermediate query results), and network bandwidth.

Our experiments show that WiseKG significantly outperforms state-of-the-art stand-alone LDF interfaces on highly demanding workloads, with increasing numbers of concurrent clients, with increasing KG sizes, and on different query shapes. We show that WiseKG's cost model improves average workload completion (reducing the number of timeouts) while also reducing resource consumption (including less CPU usage and network traffic) compared to existing interfaces.

We do however identify some shortcomings of our approach and would deserve a more dedicated analysis in future work:

- We did not perform experiments to inspect the influence of different hardware setups and mixes of clients with differing computational resources. Our future work agenda includes expanding our evaluations to a variation of hardware setups, combining realistic mixes of clients with different resources.

- We also, respectively, plan to expand our query optimizer to consider further aspects, such as additional hardware parameters, parallelism, network delays, etc. as well as to provide optimization support for additional types of queries incl., for instance, aggregation [IHPZ16] since in this work we do not model or investigate aggregation.

- We purposefully selected an established and widespread cost model [ML86] in order to assess our idea of dynamically determining the approach to process subqueries regardless of potentially complex cost models. Our proposed cost model is a kick-off to promising future refinements: indeed we plan to explore the existing plethora of different cost models from the database literature [MBK02, Gra95, GM93].

- In our cost model, we plan to look into additional parameters, for instance, to estimate the current server load: while – as we could show in our experiments – the cost model is very effective, it only observes server load as a momentary snapshot (over the last minute). We could extend this by an estimation that also takes into consideration commitments made to concurrent clients [ZSLF20] in terms of "promised" plan executions, i.e., the response to the $Plan(P)$ function to one client could also take into consideration how many commitments have been made in response to other not-yet-finished $Plan(P)$ requests to other clients. We note, however, that such an extension would also need protection against obvious DoS attacks, e.g., by "bogus" clients requesting plans that are never committed. As such, our current, inspired by R^* optimizer, is relatively simple but has significantly outperformed the state of the art.

- In addition, we note that our current considerations focus only on BGP queries. While our implementation covers also full SPARQL patterns (incl. UNION, OPTIONAL, FILTER, etc.) computed on the client side, the current approach is not dealing with multiple (named) graphs and GRAPH queries. Looking into extensions of HDT towards handling quads [FMPR18] could address this current limitation.

- Assessing the impact of ι on performance and the optimal value for ι is part of our future work. We plan to provide a detailed analysis with regard to the impact of the time quantum ι on performance and an assessment of the optimal time quantum for future work.

- The current cost model just takes the current server load as a constant factor into account, whereas a more accurate model could be expected if the server was about to take into consideration of *commitments* it has made to clients; however, in such cases, the server would need to include expiry times for such commitments, upon which expires.

- As we mentioned earlier, in our experiments, the communication component dominates the query execution time. Extensive experiments are required to verify this phenomenon in various settings.
- We verify the cost model in disk-resident settings (i.e we query the HDT files from the disk), it would be interesting to investigate the query performance in main-memory and distributed databases, and cloud computing settings which provide different challenges than our disk-resident settings.
- Our introduced cost model involves parameters that are subject to tuning by the data publishers (eg. I/O, CPU, and communication parameters). In various settings, the default values of the weight parameters could be sub-optimal [LGM+15]. In the future, it would be interesting to investigate how can we automatically adapt the cost variables according to the data publishers' environment.
- It is worth noting that we did not investigate the quality of characteristic sets cardinality estimates and how fair and accurate our cost model is in predicting the query execution time. In the future, detailed experiments are required to avoid significant misestimations that could potentially produce wrong plans.

CHAPTER 6

Smart-KG$^+$: Further Optimizations of Family-partition-based LDF

Partition-based LDF generalizes LDF interfaces, which return compressed and queryable partitions that can be used to answer several triple patterns in a single request. As a concrete implementation of partition-based LDF interfaces, we have introduced `smart-KG` and `WiseKG`, which can transfer partitions of KGs from the server to the client, reducing server resource consumption. In Chapter 4 and 5, we argue (and have experimentally demonstrated) that our approaches using families provide a reasonable trade-off of shipping sizes; still utilizing/combining other (existing) partitioning techniques could potentially provide efficient partition-based interfaces.

According to our results in Chapter 4, shipping compressed and queryable family partitions increases the server availability while achieving competitive query performance. However, there is still room for reducing the shipped KG partitions by further developing the partitioning technique. For instance, in practice, many star-shaped sub-queries include at least some bounded objects for the *rdf:type* predicate. To verify this claim, we have analyzed the real-world LSQ query log [SAH$^+$15] and found that 90% of the queries contain star-shaped patterns with at least one bounded type predicate to an object value (i.e., to a class).

Motivated by the previous phenomenon, in this chapter, we propose and formalize, and extend `smart-KG`$^+$, where we additionally introduce a graph partitioning technique named *typed family-partitioning* that benefits from this phenomenon by horizontally partitioning the families based on the classes of the entities. In addition, we propose a novel `smart-KG`$^+$ server-side partition-aware query planner to create an optimized query plan where the subqueries within a query are ordered based on the pre-computed

cardinality estimations (i.e characteristic sets). On the client-side, based on the received query plan from the server, we perform the joins locally based on implementing an asynchronous pipeline of iterators executing first the most selective iterator in order to produce the join results in an incremental fashion.

Chapter Organization. The remainder of this chapter is structured as follows:

- In Section 6.1, we introduce a possible concrete implementation of *partition-based LDF* interfaces based on extending family partitioning.
- In Section 6.2, We present a concrete implementation of partition-based LDF, `smart-KG`$^+$, that ships compressed, queryable KG partitions to distribute the processing of SPARQL queries between clients and servers.
- In Section 6.3, we detail the KG partition creation process of *(typed) family-partitioning* which extends the family partitioning technique introduced in [AFA$^+$20] to consider both predicates and classes specified in a query.
- In Section 6.4 we elaborate the query processing of `smart-KG`$^+$ and the dynamicity between clients and the server.
- In Section 6.5, we provide an empirical evaluation, and results are discussed.

  In Section 6.7, we summarize and outline the approach limitations.

6.1 Partition-based Linked Data Fragments: Typed-Family Partitioning

In Chapter 4, we have defined a *predicate family* of a subject $F(s)$ as the set of predicates related to the subject s (cf. Definition 4.1). We denote as $F(G)$ or just F, to the set of different *predicate* families in G (cf. Definition 4.2). Note that predicate families imply a partitioning usable for partition-based LDF where a partition G_i is defined for each occurring predicate family F_i (cf. Definition 4.3). We have referred to this partitioning as *family-partitioning*.

Next, the admissible queries for family-partitioning are star-shaped query patterns, i.e. BGPs composed of k triple patterns form $Q = \{(s, p_i, o_i) \mid 1 \leq i \leq k,\ s \in V \cup U, p_i \in U, o_i \in V \cup U \cup L\}$ with a single common subject s, where

$$s(G, Q) = \{G_i \in \mathcal{G} \mid pred(Q) \subseteq F_i\}$$

Obviously, for any star-shaped query, $s(G, Q)$ contains all relevant triples from G to compute the answers.

While – as we will see – family-partitioning provides a solid basis for partition-based LDF, unfortunately, family partition sizes can be significantly skewed for very popular classes

(with a large number of instances), or, respectively, very large partitions could be further subdivided by the different (sub-)classes occurring for subjects. For instance, assume common attributes $\{title, hasDirector, year, rdf{:}type\}$ for subjects of *rdf:type* Film, would also occur for each of the subclasses of Film. Intuitively, you can further subdivide each family partition "horizontally", by the different *rdf:type*s per subject. Further, we note that, based on observations of query logs for common public SPARQL query services, a large majority of user queries use bound rdf:type predicates in their queries: to back up this claim, we analyzed the real-world DBpedia LSQ [SAH+15] query log where we found out that the percentage of queries with at least one star query with a bounded *rdf:type* predicate is 88% (excluding single triple queries).

Based on these observations, we propose an extension of family-based partitioning, called *typed-family partitioning*. Assuming (without loss of generality) that the set of class URIs and predicate URIs are disjoint,[1] we can then easily extend the concept of (predicate) families to *typed-families* as follows:

$$F^{typed}(s) = F(s) \cup \{c \mid (s, rdf{:}type, c) \in G\} \tag{6.1}$$

Analogously, we extend the other notions from above, i.e., the set of typed families for a graph G:

$$F^{typed}(G) = \{F^{typed}(x) \mid x \in subj(G)\} \tag{6.2}$$

and again the notion of *typed partitions* $G^{typed}_{F_i}$ corresponding to a famlily $F_i \in F^{typed}(G)$ implies a partioning of $\mathcal{G}$ as follows:

$$\mathcal{G} = \{G^{typed}_{F_i} \mid F_i \in F^{typed}(G)\} \tag{6.3}$$

where $G^{typed}_{F_i}$ can be defined for each typed family $F_i \in F^{typed}(G)$ as

$$G^{typed}_{F_i} = \{(s, p, o) \in G \mid F^{typed}(s) = F_i\} \tag{6.4}$$

Again, we simply write G_i for $G^{typed}_{F_i}$, and finally, analogously can define

$$\sigma(G, Q) = \{G_i \in \mathcal{G} \mid pred(Q) \cup types(Q) \subseteq F_i\} \tag{6.5}$$

for again star-shaped admissible queries Q, where by $types(Q)$ we denote all (non-variable) objects of *rdf:type* triple patterns in Q.

To illustrate the previous definitions consider the KG G shown in Fig. 6.1, and the typed families shown in Fig. 6.2 Following the definition of typed-family in Eq. (6.1), the subject *s1* belongs to family F^{typed}_1, the subject *s2* belongs to family F^{typed}_2, and the subject *s3* belongs to family F^{typed}_3. Note that in predicate families, the subjects *s1* and *s2* were in

[1] Of course this does not generally hold in RDF, but we make this assumption merely to simplify notation.

```
:s1 rdf:type :Film . #t1
:s1 rdf:type :Work . #t2
:s1 :starring :o1 . #t3
:s1 :director :o2 . #t4
:s2 rdf:type :Work . #t5
:s2 :starring :o1 . #t6
:s2 :director :o3 . #t7
:s3 rdf:type :Work . #t8
:s3 :director :o4 . #t9
```

Figure 6.1: KG example

Predicate Families

$F(:s1) = \{rdf{:}type, director, starring\} = F_1$

$F(:s2) = \{rdf{:}type, director, starring\} = F_1$

$F(:s3) = \{rdf{:}type, director\} = F_2$

Predicate Families in G

$F(G) = \{F_1, F_2\}$

Partitions induced by each family

$G_{F_1} = \{t1, t2, t3, t4, t5, t6, t7\}$

$G_{F_2} = \{t8, t9\}$

Partitioning $\mathcal{G} = \{G_{F_1}, G_{F_2}\}$

Typed Families

$F^{typed}(:s1) = \{rdf{:}type, director, starring, Film, Work\} = F_1^{typed}$

$F^{typed}(:s2) = \{rdf{:}type, director, starring, Work\} = F_2^{typed}$

$F^{typed}(:s3) = \{rdf{:}type, director, Work\} = F_3^{typed}$

Predicate Families in G

$F^{typed}(G) = \{F_1^{typed}, F_2^{typed}, F_3^{typed}\}$

Partitions induced by each family

$G^{typed}_{F_1^{typed}} = \{t1, t2, t3, t4\}$

$G^{typed}_{F_2^{typed}} = \{t5, t6, t7\}$

$G^{typed}_{F_3^{typed}} = \{t8, t9\}$

Partitioning $\mathcal{G} = \{G^{typed}_{F_1}, G^{typed}_{F_2}, G^{typed}_{F_3}\}$

Figure 6.2: Predicate families and typed families for the KG shown in Fig. 6.1

the same family; this is no longer the case, as their set of classes is different. For the KG G, there are three typed families denoted $F^{typed}(G)$, i.e., F_1^{typed}, F_2^{typed} and F_3^{typed}. Lastly, each of these families induces a partition over G. For example, F_2^{typed} contains all the triples of the subject *s2*, which in this case is triples *t8* and *t9*. Lastly, the set of partitions computed for G, denoted $\mathcal{G}$ are $G_{F_1}^{typed}$, $G_{F_2}^{typed}$, and $G_{F_3}^{typed}$.

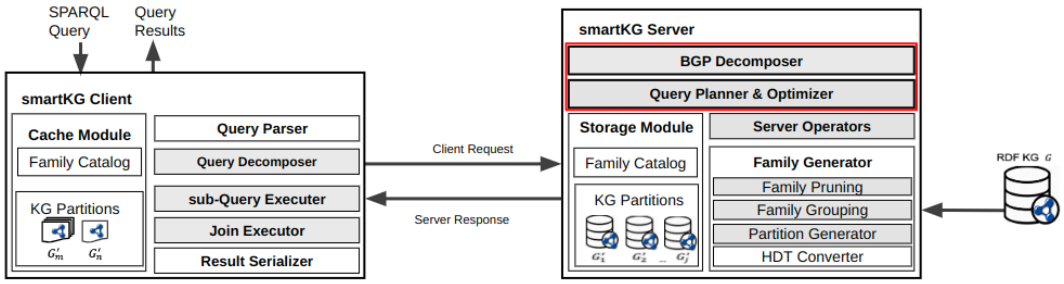


Figure 6.3: The overall architecture of the smart-KG+ client and server, wherein the modified components are denoted in red, in contrast to the corresponding elements in the original smart-KG architecture.

6.2 SMART-KG+: Design and Overview

smart-KG+ (cf. Fig. 6.3), which extends the original prototype presented in [AFA+20], combines shipping HDT compressed family partitions with the shipping of intermediate results from evaluating a given sub-(query) over the existing LDF interfaces. As such, smart-KG+ relies on both shipping intermediate results from executing single-triple patterns using a brTPF LDF interface on the server, as well as using a (typed) family-partition-based LDF interface for star-shaped subqueries (which will be evaluated on the client side, based on the shipped partition). The rest of SPARQL complex patterns other than triple or star-patterns will be evaluated on the client side.

Initially, the smart-KG+ server constructs the family-based partitions (cf. Section 6.3) for a given knowledge graph. The generated KG partitions are materialized as HDT files in the storage module together with *family catalog* that summarizes metadata information about the KG partitions including structural and statistical metadata. In addition, smart-KG+ API offers access to the KG based on two operators: one to execute a single triple pattern and the other to ship the requested partition to smart-KG+ client.

Upon receiving a BGP Q from smart-KG+ clients, the smart-KG+ server decomposes the input query into a set of o star-shaped subqueries where the server query planner devises an annotated query plan Π that decides for each pattern whether to be executed using brTPF or partition shipping. The client then evaluates the annotated query plan received from the server based on the specified subquery ordering.

As a side note, getting back to our original formalization of LDF and the fact that we do not consider "paging" (Φ) in relation to partition-based LDF: note that it would not make sense to decompose family-based partitions into chunks since chunking up the HDT-compressed partitions would require decompression.

6.3 SMART-KG+ Extending Partition Generator

We detail how the smart-KG+ server, upon loading an RDF KG, processes it into partitions per class per family, as described in Eq. (6.4) and stores those partitions as HDT compact files format.

The prior definitions carry over to *predicate-restricted typed-family partitions* and *typed-partitions*, i.e. F'^{typed} and $G'_{i,c}$ can be defined analogously, where we additionally restrict the classes by a set C'_G.

Note that, however $\mathcal{G}'$ is no longer a full cover of G, but the graph $G' = \bigcup G'_i$ only contains the "projection" of G to P'_G, with the intention that predicates other than P'_G (or, resp. classes other than C'_G) are delegated to brTPF.

Serving predicate restricted families allows a smart-KG+ publisher to select P'_G (and C'_G) depending on (i) the cardinality of the predicates (i.e. the number of occurrences in the graph) and (ii) the importance of predicates (and combinations) in actual query workloads. We will describe a concrete method to pick P'_G (and C'_G) based on the cardinality of predicates and classes in Section 6.3.1.

6.3.1 Family Pruning

As we discussed earlier, the cost of fully materializing the partitions generated from *all* potential merges (intersections) of all families in G could prohibitive.

We extend the family pruning strategy introduced in Section 4.2.1.3 for restricting the number of materialized partitions, where we (i) restrict considered predicates in P'_G based on their cardinality (cf. Eq. 4.6), (ii) avoid the creation of small families that deviate only slightly from another overlapping, "core" families (cf. Eq. 4.7), and (iii) avoid materialization of families over a certain size (cf. Eq. 4.8). In the following, we extend the family pruning strategy where we (iv) restrict considered classes based on their cardinality in generating typed-families,

(iv) Restrict classes cardinality based on cardinality to generate typed-families. As discussed when introducing typed partitions, `rdf:type` is a natural horizontal partitioner for predicate families, which plays an essential role in reducing the size of the shipped families; plus, as also mentioned above, `rdf:type` is a heavy hitter in real-world queries, since it is a frequently used predicate in log queries as shown in Table 6.2.

Therefore, similar to issue (i), the cardinality of classes contributes to the number and size of the shipped partitions: firstly, rare classes occurring as `rdf:type` objects in triple patterns are by nature selective: such triple patterns are better handled through a TPF/brTPF/SPF call without shipping a *typed* family; secondly, frequent classes can be potentially present in many of the families (for instance `owl:Thing`) and, in practice, are rarely used in queries.[2]

[2]For instance `foaf:Document` is a large class but mentioned in only 68 queries in LSQ query log

We address the aforementioned issues, similar to issue (i), by excluding these classes, and maintaining minimum ($\tau_{class_{low}}$) and maximum ($\tau_{class_{high}}$) thresholds for the percentage of triples per class. We rely on these two thresholds to define the set of classes C'_G for restricting the created typed partitions:

$$C'_G = \{c \in types(G) \mid \tau_{class_{low}} \leq \frac{|(s, rdf{:}type, c) \in G|}{|G|} \leq \tau_{class_{high}}\} \tag{6.6}$$

Finally, we avoid the materialization of overly large (e.g. hundreds of millions of triples in DBpedia) merged partitions G_I with size G_I above a threshold α_t, which limits the size of the materialized merged partitions.

In order to only take core families into account for the creation of partitions, and limit merged families to sizes below α_t, it is sufficient to modify Equation (4.8) as follows:

$$\mathcal{G}_{serv} = \underbrace{\left\{ G'_{\mu(f)} \;\middle|\; \begin{array}{l} f \in dom(\mu) \wedge \\ \mu(f) \cap I_{core} \neq \emptyset \wedge \\ \sum_{i \in \mu(f)} |G'_i| \leq \alpha_t \end{array} \right\}}_{\text{Merged partitions}} \cup \underbrace{\{G_{\{i\}} | F'_i \in F'\}}_{\text{Non-merged partitions}} \cup \underbrace{\{G_{i,class} | class \in C'_G, F'_i \cup \{class\} \in F^{typed}\}}_{\text{Typed partitions}} \tag{6.7}$$

In Equation (6.7), line 2 addresses issue (ii)[3] and line 3 addresses issue (iii)[4]. The second part ensures that, despite pruning, the non-merged partitions of families in F' remain being served. While the last part ensures serving the typed-families. In practice, the `smart-KG`$^+$ materializes the partitions in $\mathcal{G}_{serv}$ as HDT files.

6.4 SMART-KG$^+$: Query Processing

In this section, we detail how the `smart-KG`$^+$ server and client work together to process SPARQL queries. In particular, we describe how the query processing is performed on the server-side and on the client-side.

6.4.1 SMART-KG$^+$ Server

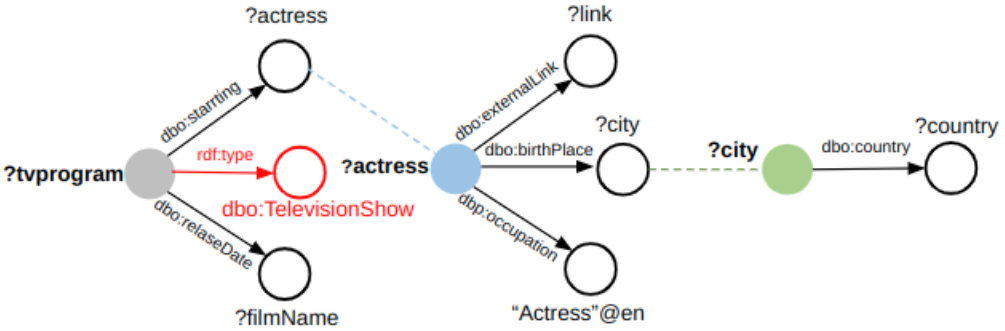
In this section, we describe how `smart-KG`$^+$ server query planner creates a query execution plan for a submitted BGP. In addition, we detail how `smart-KG`$^+$ server enables the clients to access the partitions constructed by the partition generator described in Section 6.3 and to evaluate single triple patterns using brTPF.

[3] since $Subj(G'_i) \cap Subj(G'_j) = \emptyset$ for all base families $F'_i, F'_j \in F'$, by construction it holds that $|Subj(G'_I)| = \sum_{i \in I} |Subj(G'_I)|$

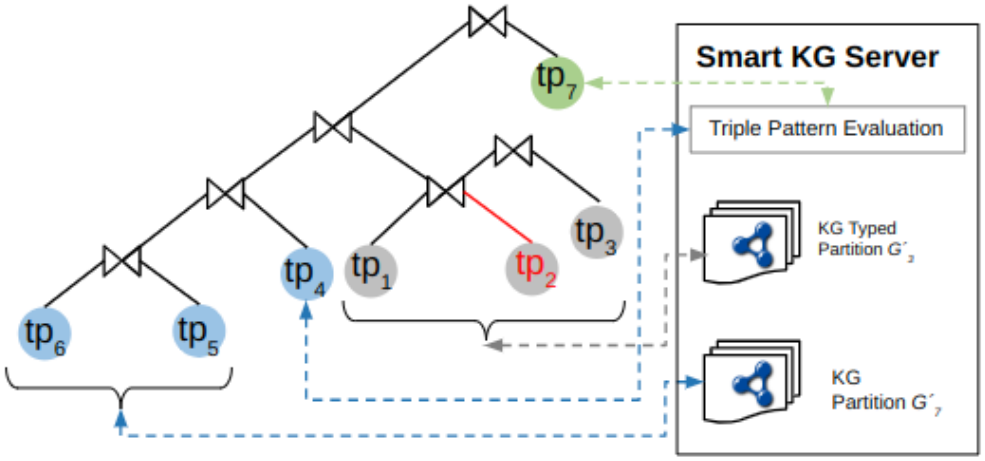
[4] since $|G'_I| = \sum_{i \in I} |G'_i|$

```
SELECT * WHERE {
    ?tvprogram dbo:starring  ?actress .          # tp1  544,110 matches
    ?tvprogram dbo:releaseDate  ?releaseDate .   # tp2  155,199 matches
    ?tvprogram rdf:type dbo:TelevisionShow.      # tp3  85,660 matches
    ?actress  dbo:wikiPageExternalLink  ?link .  # tp4  9,643,439 matches
    ?actress  dbo:birthPlace  ?city .            # tp5  1,469,160 matches
    ?actress  dbp:occupation        "Actress"@en.  # tp6  18,861 matches
    ?city  dbo:country  ?country.                # tp7   789,261 matches
}
```

(a) Select all actresses, their corresponding TV programs of the "Televisionshow" type, and their birthplace information, with an additional triple pattern highlighted in red when compared to the example query presented in Chapter 4.



(b) Star-shaped query decomposition.



(c) Server Query Plan

Figure 6.4: Example of processing a SPARQL query with the `smart-KG+` client.

6.4.1.1 Query Decomposer

First, smart-KG$^+$ splits parsed Basic Graph Patterns (BGPs) into *stars* as follows: given a BGP Q, with subjects $subj(Q)$, a decomposition $\mathcal{Q} = \{Q_s \mid s \in subj(Q)\}$ of Q is a set of star-shaped BGPs Q_s such that $Q = \bigcup_{s \in subj(Q)} Q_s$ and:

$$Q_s = \{tp \in Q \mid tp = (s, p, o)\} \tag{6.8}$$

Analogous to graphs, we can also associate a family to each star query Q_s:

$$F(Q_s) = \{p \mid \exists o : (s, p, o) \in Q_s, p \in U\} \tag{6.9}$$

Given the SPARQL query in Fig. 6.4a, the BGP is decomposed into $\mathcal{Q} = \{Q_{\texttt{?tvprogram}}, Q_{\texttt{?actress}}, Q_{\texttt{?city}}\}$ around the three subjects (cf. Fig. 6.4b). Each of the star families $F(Q_s)$ that can be mapped to existing predicate families in $dom(\mu_G)$ on the server has a non-empty answer. For example, $Q_{\texttt{?tvprogram}} = \{$(?tvprogram, dbo:starring, ?actress), (?tvprogram, dbo:releaseDate, ?releaseDate), (?tvprogram, rdf:type, dbo:TelevisionShow)$\}$ has $F(Q_{\texttt{?tvprogram}}) = \{$dbo:starring, dbo:releaseDate, rdf:type$\}$; based on the decomposition $\mathcal{Q}$, smart-KG$^+$ server query planner generates a query plan to the input query. Note that $Q_{\texttt{?tvprogram}}$ contains rdf:type which will be distinguished by the query planner and optimizer while devising the query plan.

6.4.1.2 Shipping-based Query Planner & Optimizer

In smart-KG$^+$, the query planner and the optimizer are executed at the server-side to provide more efficient query plans based on pre-computed characteristic set cardinality estimations and the server's partition metadata. When the smart-KG$^+$ server receives a request $Plan(Q)$ for a BGP Q from a client, the server query planner devises an *annotated query plan* to specify which interfaces are used per subquery. The interfaces are denoted with superscripts to represent triple pattern shipping using TPF and partition shipping using SKG to describe the interfaces as $Q_s^{interface}$.

Given $\mathcal{Q}$, P'_G, and C'_G as input, the query optimizer devises a query plan Π_Q where the resp. algorithm to compute $Plan(Q, P'_G, C'_G)$ is shown in Alg. 6.1. If the result is trivially empty (lines 4-5), it returns an empty plan. Then, the optimizer finds the star-subquery Q_{s_i} with the lowest cardinality estimation using the function $card(Q_s, \Pi_Q)$ (line 6); in our running example, this would order $Q_{?actress}$, followed by $Q_{?films}$, and lastly the triple pattern in $Q_{?city}$.

Next, Q_{s_i} is annotated with the corresponding controls that represent the evaluation of the subquery: SKG control and TPF control. Therefore, the optimizer characterizes each Q_{s_i} to decide whether to use partition shipping or triple pattern (for parts) of Q_{s_i}, as follows:

Partition Shipping. Shipping relevant partitions to evaluate a star $Q_s \in \mathcal{Q}$ requires considering the materialized partitions at the server. Since graph partitions are generated

based on the pruned families (cf. Sec. 6.3), only stars with $F(Q_s) \subseteq P'_G$ can be fully evaluated by served partitions. Therefore, the optimizer partitions each $Q_{s_i} \in \mathcal{Q}$ into the disjoint sets Q'_{s_i} and Q''_{s_i}, where Q'_{s_i} is the part of the star that can potentially be evaluated over the served partitions (line 7), whereas the remaining triple patterns in Q''_{s_i} are delegated to brTPF requests (line 8). Note that Q''_{s_i} also includes triple patterns with predicate variables, i.e., $p \in V$. Then, the optimizer checks (lines 9–12) for each triple pattern tp with `rdf:type` predicate whether the class o belongs to the restricted set of classes C'_G. Note that this also captures the case with $o \in V$. In the negative case, there is no materialized typed-family partition to serve Q_{s_i}, therefore, this tp will be pushed to Q''_s to be evaluated using brTPF (line 11). Then, the optimizer considers heuristics that partition shipping is only followed if $|Q'_{s_i}| > 1$ where Q'_{s_i} is annotated with the respective control for partition shipping SKG (lines 15–16). As in practice, the transfer of graph partitions to resolve a single triple pattern usually takes longer than delegating to a brTPF request directly (line 18).

Triple Pattern Shipping. Q_s with a single triple pattern, triple patterns with infrequent predicates, and triple patterns with predicate variables will be annotated with TPF control (lines 17). These subqueries will be eventually evaluated using a brTPF request to the server.

Lastly, the optimizer reorders the triple patterns with the function *reorder* in the sub-plans based on their cardinality estimations; this allows for an efficient evaluation at the client side. Then, the optimizer attaches the sub-plans with the *append* function to build the final plan Π_Q. The resulting query plan Π_Q comprises sub-plans annotated with the corresponding shipping strategy.We describe a full annotated query plan as sequences of patterns being interpreted as left-linear query plans, that is, we write query plans that evaluate patterns as permutations of the decomposed stars in $\mathcal{Q}$. For our example shown in Fig. 4.4a, the annotated plan could be written as $\Pi = (Q'^{SKG}_{?actress}, Q''^{TPF}_{?actress}, Q^{SKG}_{?film}, Q^{TPF}_{?city})$, describing an execution plan at the level of joining star patterns as follows: $(((Q'_{?actress} \bowtie Q''_{?actress}) \bowtie Q_{?film}) \bowtie Q_{?city})$. Fig. 4.4c shows the shipping strategies for each sub-plan from our example. The query optimizer first starts with the subquery $Q_{?actress}$ which is the most selective subquery. For $Q_{?actress}$, the optimizer creates $Q'^{SKG}_{?actress} \bowtie Q''^{TPF}_{?actress}$, as the triple pattern tp_4 in $Q_{?actress}$ is evaluated using triple pattern shipping as the optimizer determined that the predicate `dbo:wikiPageExternalLink` is not in P'_G. Next, the query optimizer evaluates $Q_{?film}$ via partition shipping on the client based on a family-based partition. Finally, $Q_{?city}$ is added to be executed as a single triple pattern using brTPF.

6.4.1.3 Server Operators

The `smart-KG`$^+$ server provides operators to ship partitions and their metadata, or to respond to brTPF requests. These operators are defined in the following interface calls to access a KG G:

Algorithm 6.1: Query Optimizer and Planner: *optimizePlan*

Input: Star-shaped query decomposition $\mathcal{Q}$, P'_G, C'_G
Output: Π_Q an annotated query plan for $\mathcal{Q}$

$\Pi_Q \leftarrow ()$
while $\mathcal{Q} \neq \emptyset$ **do**
  for $Q_s \in \mathcal{Q}$ **do**
    if $card(Q_s, \Pi_Q) = 0$ **then**
      return ()
    end
  end
  $Q_{s_i} \leftarrow Q_s \in \mathcal{Q}$ such that $card(Q_s, \Pi_Q) \leq card(Q_{s_j}, \Pi_Q)$ for all $Q_{s_j} \in \mathcal{Q}$
  $Q'_{s_i} \leftarrow \{(s_i, p, o) \in Q_{s_i} \mid p \in P'_G\}$
  $Q''_{s_i} \leftarrow Q_{s_i} \setminus Q'_{s_i}$
  `// Check if there exists family-typed partitions for classes in` Q'_{s_i}
  for $tp = (s_i, rdf{:}type, o) \in Q'_{s_i}$ **do**
    if $o \notin C'_G$ **then**
      $Q''_{s_i} \leftarrow Q''_{s_i} \cup \{tp\}$
      $Q'_{s_i} \leftarrow Q'_{s_i} \setminus \{tp\}$
    end
  end
  `// Re-order triple patterns within subqueries`
  $Q'_{s_i} \leftarrow reorder(Q'_{s_i})$
  $Q''_{s_i} \leftarrow reorder(Q''_{s_i})$
  `// Annotate plan with interface`
  if $|Q'_{s_i}| > 1$ **then**
    $\Pi_Q \leftarrow append(\Pi_Q, Q'^{SKG}_{s_i})$`// Evaluate plan using partitions`
  end
  else
    $\Pi_Q \leftarrow append(\Pi_Q, Q'^{TPF}_{s_i})$`// Evaluate plan using TPF`
  end
  `// Evaluate tps with no materalized partitions using TPF`
  $\Pi_Q \leftarrow append(\Pi_Q, Q''^{TPF}_{s_i})$
  $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{Q_{s_i}\}$
end
return (Π_Q)

- A brTPF LDF API control $TPF(Q_s, \Omega)$ returning $\sigma_{brTPF}(G, Q_s, \Omega) = s(G, Q_s, \Omega)$, that retrieves the answers for a single triple pattern Q_s while taking into consideration the attached bindings Ω, i.e., the `smart-KG`$^+$ server returns the triples from G that match Q_s based on brTPF requests.

- A SKG LDF API control $SKG(Q_s, \emptyset)$ that handles star-shaped queries Q_s and returns $\sigma_{SKG}(G, Q_s, \emptyset) = \sigma(G, Q)$, i.e., the set of typed-family partitions if exists, otherwise the family partitions, of which $[\![Q_s]\!]_G$ can be computed and joined on the client-side.
- $Plan(Q)$ to create a query plan for the received BGP Q. Unlike the initial `smart-KG+` prototype [AFA+20], where the query plan was inaccurately created on the client-side, we propose shifting the query execution planning from the client to the server to compute better query plans as the server has access to more accurate cardinality estimations to determine the order of stars and triple patterns.

6.4.2 SMART-KG$^+$ Client

The primary focus of this work is on evaluating BGPs as the essential retrieval functionality of the SPARQL query language. However, our introduced interface is able to process a full SPARQL query including operators such as `UNION` and `OPTIONAL`, `FILTER`, etc., which are all evaluated locally on the client-side. Herein, we introduce the general approach for processing a SPARQL query, as follows:

1. Upon receiving a SPARQL query, the *query parser* translates the input query string into the corresponding SPARQL algebra expressions.
2. Initially, the client sends a request $Plan(Q)$ to retrieve from the server an optimized query execution plan Π_Q for the extracted BGP Q.
3. The *query executor* evaluates the received plan and iteratively combines the results using a dynamic pipeline of iterators, following brTPF [HA16], where each iterator deals with a certain annotated subquery Q_s^c that request a partition or performs a brTPF request.
4. The *results serializer* translates the locally joined results into the specified format. Note that the downloaded partitions from the `smart-KG+` server during query evaluation can be locally stored in the *family cache* to be reused in the upcoming queries.

We describe in detail the algorithms that implement the query executor in a recursive manner in Alg. 6.2 and Alg. 4.2. The function *evalPlan* recursively evaluates the received plan Π by traversing the left-tree of sub-plans (cf. Alg. 6.2). The query executor initially evaluates the first and most selective subquery Q_{s1}^{c1} (line 1). Then, the algorithm traverses the rest of the plan (lines 2-6). The base case is when the plan is associated with a single star pattern Q_s^c (line 2). In this case, the executor evaluates the star using $eval_c(Q_s^c, \Omega')$ while considering the intermediate results from earlier subqueries Ω' (for details, cf. Alg. 6.3). Otherwise, the query executor will recursively call the evaluation of the remaining subtree and join them with the set of bindings retrieved from the previous calls (line 5). The final output of the query executor is the query result set Ω of a given

Algorithm 6.2: Query Executor: *evalPlan*

Input:
$\Pi = (Q_{s_1}^{c_1}, \ldots, Q_{s_n}^{c_n})$ // an execution plan for a BGP query with stars $Q_{s_1}^{c_1}, \ldots, Q_{s_n}^{c_n}$;
Output: Ω // set of solution mappings

1 $\Omega \leftarrow eval_c(Q_{s_1}^{c_1}, \emptyset)$
2 **if** $|(Q_{s_2}^{c_2}, \ldots, Q_{s_n}^{c_n})| = 1$ **then**
3 $\quad \Omega \leftarrow \Omega \bowtie eval_c(Q_{s_n}^{c_n}, \Omega)$
4 **end**
5 **else if** $|(Q_{s_2}^{c_2}, \ldots, Q_{s_n}^{c_n})| > 1$ **then**
6 $\quad \Omega \leftarrow \Omega \bowtie evalPlan(Q_{s_2}^{c_2}, \ldots, Q_{s_n}^{c_n})$
7 **end**
8 **return** Ω

plan (line 6). In practice, the executor implements an iterator to push intermediate results of evaluating one sub-plan to the next operator in the plan. This allows the smart-KG$^+$ client to incrementally stream query results once computed.

Alg. 6.3 presents the function $eval_c(Q_s^c, \Omega')$, which calls the corresponding interface (i.e. the respective smart-KG$^+$ server operators for the shipping strategy determined by the server query plan). The first case $c = SKG$ is to evaluate a star pattern using a shipped partition on the client-side. The second case $c = TPF$ involves calling the brTPF interface which directly returns the result bindings. In the following, we explain the two cases in detail:

Case SKG. Each sub-plan Q_s^{SKG} is evaluated (cf. Alg. 6.3, lines 2–5) by retrieving HDT partitions using the server operator for partition shipping $SKG(Q_s, \emptyset)$. This operation returns a set of partitions G^* which is either a set of family-based partitions or a set of typed-family partitions (cf. Alg. 6.3, line 2). The query executor evaluates each triple pattern tp of the star pattern Q_s^{SKG} (lines 4–5) over the partitions (using the SPARQL algebra union operator when several partitions are retrieved). The results of each tp are joined to produce the final results of the star pattern.

Case: TPF. Each sub-plan Q_s^{TPF} involves a single triple pattern which is executed by calling the server operator *TPF(tp,* Ω'*)* which is a brTPF interface that directly returns the result bindings (cf. Alg. 6.3, lines 6-7).

6.4.3 Detailed Example:

In this section, we demonstrate a full example to explain the evaluation of the SPARQL query based on our introduced approach smart-KG. In this example, we elaborate family-based partitioning and SPARQL query evaluation on our RDF graph from Example 7 inspired *Friends* series:

Algorithm 6.3: Query Executor: $eval_c$

Input:
Q_s^c // A decomposed pattern and it is annotated execution plan;
Ω' // a set of binding if available
Output: Ω a set of solution mappings

```
if c = SKG then
    G* = SKG(Q_s^c, ∅)
    Ω ← {ω_∅}
    for tp ∈ Q_s^c do
        Ω ← Ω ⋈ ⋃_{G_j ∈ G*} [[tp]]_{G_j}
    end
end
else if c = TPF then
    Ω ← TPF(Q_s^c, Ω')
end
return Ω
```

Creation of Family-based partitioning for Friends RDF graph. In this example, we materialize family-based partitions based on three different settings, as follows:

- Setting 1: we materialize all families as well as all possible merges of families (i.e. the pruning step is not applied). In our example, it is feasible to materialize all partitions given the fact that our example graph is small of size $|G| = 79$ triples and $|P_G| = 16$ predicates. For this purpose, we fix the pruning parameters as follows: we set $\tau_l = 0$ and $\tau_h = 1.0$ so that we include all predicates including both infrequent and heavy hitters. We set $\tau_{class_{low}} = 0$ and $\tau_{class_{high}} = 1.0$ so that we include all classes in the partitioning process. Likewise, we set $\alpha_s = 0$ to include all families in the grouping step and $\alpha_t = 1.0$ to materialize all possible family merges.

  This setting will generate $|\mathcal{G}_{serv}| = 21$ materialized partitions based on the following set of families:

  - F_1={`dbo:portrayer`, `dbp:children`, `dbp:family`, `dbp:gender`, `dbp:occupation`, `dbp:religion`, `rdf:type`, `rdfs:label`} which is an *original* family that generates a partition G_1 where $|G_1| = 9$ triples with a set of subjects $subj(G_1)$ of size $|subj(G_1)| = 1$ subjects. This subject represents the fictional Chandler since she is the only member of the graph that have the combination of predicates in F_1. In this family, the only entity is Chandler Bing, and he belongs to the type 'FictionalCharacter', so the family is not partitioned based on the type predicate.
  - F_2={`dbo:birthDate`, `dbo:education`, `dbp:occupation`, `dbp:spouse`, `rdf:type`, `rdfs:label`} which is an *original* family that

generates a partition G_2 where $|G_2| = 18$ triples with a set of subjects $subj(G_2)$ of size $|subj(G_2)| = 3$ subjects. These subjects represent two actresses Courteney Cox and Jennifer Aniston and one actor David Schwimmer as they share the set of predicates in F_2. In this family, all entities belong to a single type 'Person' so the family is not partitioned based on the type predicate.

- F_3={`dbo:almaMater`, `dbo:birthDate`, `dbp:occupation`, `dbo:spouse`, `rdf:type`, `rdfs:label`} which is an *original* family that generates a partition G_3 where $|G_3| = 6$ triples with a set of $subj(G_3)$ of size $|subj(G_3)| = 1$ subject. This subject represents the actress Lisa Kudrow. In this family, the only entity is Lisa Kudrow belongs to a single type 'Person' so the family is not partitioned based on the type predicate.
- F_4={`dbo:birthDate`, `dbp:occupation`, `dbp:spouse`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_4 where $|G_4| = 30$ triples with a set of $subj(G_4)$ of size $|subj(G_4)| = 7$ subjects. These subjects represent the two actors Matt LeBlanc and David Schwimmer as well as three actresses Courteney Cox, Jennifer Aniston, and Lisa Kudrow in addition to the fictional character Ross Geller as they share the set of predicates in F_4. In this case, we will create two typed families, F_1^{typed} and F_2^{typed}, that generate $G_{4,FictionalCharacter}$ and $G_{4,Person}$, respectively, for entities of those types.
- F_5={`dbo:birthDate`, `dbp:occupation`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_5 where $|G_5| = 28$ triples with a set of $subj(G_5)$ of size $|subj(G_5)| = 7$ subjects. These subjects represent the two actors Matthew Perry and David Schwimmer as well as three actresses Courteney Cox, Jennifer Aniston, and Lisa Kudrow in addition to the fictional character Ross Geller as they share the set of predicates in F_5. In this case, we will create two typed families, F_3^{typed} and F_4^{typed}, that generate $G_{5,FictionalCharacter}$ and $G_{5,Person}$, respectively, for entities of those types.
- F_6={`dbo:portrayer`, `dbp:occupation`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_6 where $|G_6| = 25$ triples with a set of $subj(G_6)$ of size $|subj(G_6)| = 6$ subjects. These subjects represent the 6 fictional characters of the show including Ross, Monica, Chandler, Joey, Phoebe, and Rachel as they share the set of predicates in F_6. In this family, all entities belong to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.
- F_7={`dbo:alias`, `dbo:portrayer`, `dbo:affiliation`, `dbp:nationality`, `dbp:occupation`, `dbp:religion`, `rdf:type`, `rdfs:label`} which is an *original* family that generates a partition G_7 where $|G_7| = 8$ triples with a set of $subj(G_7)$ of size $|subj(G_7)| = 1$ subject. This subject represents the fictional character Joey Tribbiani. In this family, there

is a single entity that belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.

- F_8={`dbo:alias, dbo:portrayer, dbp:children, dbp:occupation, dbp:religion, dbp:spouse, rdf:type, rdfs:label`} which is an *original* family that generates a partition G_8 where $|G_8| = 8$ triples with a set of $subj(G_8)$ of size $|subj(G_8)| = 1$ subject. This subject represents the fictional character Monica Geller. In this family, there is a single entity that belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.
- F_9={`dbo:portrayer, dbp:family, dbp:gender, dbp:nationality, dbp:occupation, rdf:type, rdfs:label`} which is an *original* family that generates a partition G_9 where $|G_9| = 8$ triples with a set of $subj(G_9)$ of size $|subj(G_9)| = 1$ subject. This subject represents the fictional character Phoebe Buffay. In this family, there is a single entity that belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.
- F_{10}={`dbo:birthDate, dbo:portrayer, dbp:children, dbp:occupation, dbp:religion, dbp:spouse, dbp:title, rdf:type, rdfs:label`}
which is an *original* family that generates a partition G_{10} where $|G_{10}| = 9$ triples with a set of $subj(G_{10})$ of size $|subj(G_{10})| = 1$ subject. This subject represents the fictional character Ross Geller. In this family, there is a single entity which belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.
- F_{11}={`dbp:occupation, rdf:type, rdfs:label`} which is a *grouped* family that generates a partition G_{11} where $|G_{11}| = 37$ triples with a set of $subj(G_{11})$ of size $|subj(G_{11})| = 6$ subject. These are all unique subjects in the Friends graph as all actors, actresses, and fictional characters have the set of predicates in F_{11}. In this case, we will create two typed families, F_5^{typed} and F_6^{typed}, that generate $G_{11,FictionalCharacter}$ and $G_{11,Person}$, respectively, for entities of those types.
- F_{12}={`dbo:portrayer, dbp:occupation, dbp:religion, rdf:type, rdfs:label`} which is a *grouped* family that generates a partition G_{12} where $|G_{12}| = 20$ triples with a set of $subj(G_{12})$ of size $|subj(G_{12})| = 4$ subjects. These subjects actually represent the 4 fictional characters of the show including Ross, Monica, Chandler, and Joey as they share the set of predicates in F_{12}. In this family, there is a single entity which belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.
- F_{13}={`dbo:portrayer, dbp:children, dbp:religion, dbp:occupation, rdf:type, rdfs:label`} which is a *grouped* family that generates a partition G_{13} where $|G_{13}| = 18$ triples with a set of $subj(G_{13})$

of size $|subj(G_{13})| = 3$ subjects. These subjects represent the 3 fictional characters of the show including Ross, Monica, and Chandler as they share the set of predicates in F_{13}. In this family, there is a single entity which belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.

- F_{14}={`dbo:portrayer`, `dbp:family`, `dbp:gender`, `dbp:occupation`, `rdf:type`, `rdfs:label`}
which is a *grouped* family that generates a partition G_1 where $|G_{14}| = 14$ triples with a set of $subj(G_{14})$ of size $|subj(G_{14})| = 2$ subjects. These subjects represent the 2 fictional characters Chandler and Phoebe as they share the set of predicates in F_{14}. In this family, there is a single entity that belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.

- F_{15}={`dbp:occupation`, `dbp:spouse`, `rdf:type`, `rdfs:label`}
which is a *grouped* family that generates a partition G_{15} where $|G_{15}| = 28$ triples with a set of $subj(G_{15})$ of size $|subj(G_{15})| = 7$ subjects. These subjects represent the two actors Matt LeBlanc and David Schwimmer as well as three actresses Courteney Cox, Jennifer Aniston, and Lisa Kudrow in addition to the fictional character Ross Geller as they share the set of predicates in F_{15}. In this family, there is a single entity which belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.

- F_{16}={`dbo:alias`, `dbo:portrayer`, `dbp:occupation`, `dbp:religion`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_{16} where $|G_{16}| = 12$ triples with a set of $subj(G_{16})$ of size $|subj(G_{16})| = 2$ subjects. These subjects represent the two fictional characters Joey and Monica. In this family, there is a single entity which belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.

- F_{17}={`dbo:portrayer`, `dbp:nationality`, `dbp:occupation`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_{17} where $|G_{17}| = 10$ triples with a set of $subj(G_{17})$ of size $|subj(G_{17})| = 2$ subjects. These subjects represent the two fictional characters Joey and Phoebe. In this family, there is a single entity which belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.

- F_{18}={`dbo:portrayer`, `dbp:children`, `dbp:occupation`, `dbp:religion`, `dbp:spouse`, `rdf:type`, `rdfs:label`} which is a *grouped* family that generates a partition G_{18} where $|G_{18}| = 14$ triples with a set of $subj(G_{18})$ of size $|subj(G_{18})| = 2$ subjects. These subjects represent the two fictional characters, Monica and Ross. In this family, there is a single

entity that belongs to a single type 'FictionalCharacter' so the family is not partitioned based on the type predicate.

In Setting 1, we materialize all possible partitions $G_{serv} = \{G_{1,FictionalCharacter}, ..., G_{4,FictionalCharacter}, G_{4,Person}, ..., G_{18,FictionalCharacter}\}$.

- Settings 2 and Settings 3: All entities in each of the materialized partitions belong to exactly one of the two types, 'Person' or 'FictionalCharacter', so the result of family-based partitioning is equivalent to typed-family partitioning (see Section 4.3.1).

SPARQL Query Evaluations on Friends RDF graph. In the following, we show the evaluation of SPARQL query examples based on the materialized partitions according to the aforementioned settings:

Query 1: retrieve the list of characters featured in TV shows, including their respective occupations and information regarding the actors who portray them. Specifically, retrieve the birthdate and actual educational background of the actors in real life. This query Q can be written, as follows:

```
SELECT * WHERE {
    ?character  dbo: portrayer  ? portrayer .
    ?character  dbo:occupation ?occupation.
    ?character  rdf : type  dbo: FictionalCharacter .
    ? portrayer  dbo:birthDate  ?date.
    ? portrayer  dbo:education  ?education.
}
```

First, the query decomposer splits the BGP into two star-shaped sub-queries as follows: $\mathcal{Q} = \{Q_{?character}, Q_{?portrayer}\}$. Second, `smart-KG` query planner devises a plan where both of the decomposed stars can be fully evaluated by the served partitions since $F(Q_{?character}) \subseteq P'_{core}$ and $F(Q_{?portrayer}) \subseteq P'_{core}$ in all of the aforementioned partitioning settings. This left-linear plan is written as $\Pi = \{Plan(\Pi_{?portrayer}, \text{P-S}), Plan(\Pi_{?character}, \text{P-S})\}$.

In the following, we detail the evaluation of the query plan Π using the query executor(cf. Algo. 6.3), while considering the different partitioning settings:

- Case Setting 1: the query executor starts with evaluating $Plan(\Pi_{?portrayer}, \text{P-S})$ by shipping the partition G_2 based on the original family F_2. Note that F_2 is the only family that contains the set of predicates of $Q_{?portrayer}$ in G_{serv}. The size of the intermediate results of evaluating $Q_{?portrayer}$ triple patterns on G_2 is $|\Omega_{?portrayer}| = 18$. Then, The query executor evaluates $Plan(\Pi_{?character}, \text{P-S})$ by determining the relevant served partition for $\Pi_{?character}$ (line 3) which is a single partition G_6 based on the grouped family F_6 to resolve the star query $Q_{?character}$. The query executor evaluates each triple pattern tp_i in $Q_{?character}$ on G_6 (line 6) generating the following intermediate results $\Omega_{?character}$ that is of a size equal to $|\Omega_{?character}| = |G_6| = 25$. Note that the shipped partition contains only fictional

characters. However, if we modify our example query to include *(?character rdf:type dbo:Person)* instead of *(?character rdf:type dbo:FictionalCharacter)*, the query optimizer will not need to ship any partition from the typed family partitioning, as we can be certain that the result will be empty due to the known fact that partition G_1 contains only Fictional characters and no person entities. Finally, we join the solution mappings of the two-star sub-queries to compute the final solution mapping (line 14), as $\Omega_Q = \Omega_{?portrayer} \bowtie \Omega_{?character}$ and the result is in the following:

$\textsc{Select}_*(\Omega_Q) = \{\{character \rightarrow$ dbr:Rachel_Green,
$portrayer \rightarrow$ dbr:Jennifer_Aniston,
$occupation \rightarrow$ dbr:Louis_Vuitton, $date \rightarrow$ "1969 − 02 − 11",
$education \rightarrow$ dbr:Fiorello_H._LaGuardia_High_School$\}$,
$\{character \rightarrow$ dbr:Monica_Geller, $portrayer \rightarrow$ dbr:Courteney_Cox,
$occupation \rightarrow$ dbr:Chef, $date \rightarrow$ "1964 − 06 − 15",
$education \rightarrow$ dbr:Mount_Vernon_Seminary_and_College$\}$,
$\{character \rightarrow$ dbr:Ross:Geller, $portrayer \rightarrow$ dbr:David_Schwimmer,
$occupation \rightarrow$ dbr:Chef, $date \rightarrow$ "1966 − 11 − 02",
$education \rightarrow$ dbr:Northwestern_University$\}\}$.

6.4.4 Proof of SMART-KG$^+$ Correctness

The evaluation of SPARQL BGP queries with `smart-KG+`, as stated in the following proposition.

Proposition 2. *The result of evaluating a BGP Q over an RDF graph G with `smart-KG+`, denoted smart-eval(Q, G), is correct w.r.t. the semantics of the SPARQL language, i.e., smart-eval$(Q, G) = [\![Q]\!]_G$.*

The proof of the proposition is in the following:

Proof. For this proof, we first show that the `smart-KG+` query decomposer and planner are correct. By construction, the query decomposer is correct, as the combination of the star-shaped queries Q_s corresponds to the original Q (cf. Eq. 4.9). Furthermore, the tasks of the query planner are two-fold. First, the optimizer devises an ordering of the star-shaped queries Q_i and the triple patterns within Q_i (Alg. 6.1, lines 3–7). This first task ensures that the plans are correct since the join operator is commutative and associative in the SPARQL algebra [SML10]. The second task is to partition Q_i into subsets Q_i' and Q_i'' to be evaluated using the TPF or the SKG APIs (Alg. 6.1, lines 8–13). In the second task, it is easy to see that $Q_i = Q_i' \cup Q_i''$ and that $Q_i' \cap Q_i'' = \emptyset$, i.e., all triple patterns of the star-shaped query Q_i are evaluated once either using the TPF or SKG APIs. Lastly, since all stars in the input decomposition $\mathcal{Q}$ are processed in Alg. 6.1, the produced plans are correct.

Now we proceed to show that the execution of plans (cf. Alg. 6.3) is also correct. For this proof, we assume that the server operators (i.e., the SKG API and the LDF API) are implemented correctly. By contradiction, let us assume that *smart-eval*$(Q, G) \neq [\![Q]\!]_G$. We distinguish three cases based on the shipping strategy used for evaluating Q.

(i) Q is evaluated with Partition Shipping. For this case, we assume the correct implementation of the join operator, therefore, it is sufficient to prove this case when Q is composed of a pair of triple patterns with variable-free-predicate p and $p \in P'_G$. With partition shipping, the evaluation of Q is carried out against the set of corresponding partitions obtained with $SKG(Q, \emptyset)$. After applying the server operators, the query executor obtains the set of relevant partitions $G^* \subseteq \mathcal{G}_{serv}$ for Q (Alg. 6.3, line 2). Next, we consider two sub-cases. In the first sub-case, we have that *smart-eval*$(Q, G) \subset [\![Q]\!]_G$, i.e., there exists an RDF triple $t \in G$ with predicate p such that $t \notin \bigcup_{G_j \in G^*} G_j$. Therefore, the partitions in $\mathcal{G}_{serv}$ are created incorrectly, which contradicts Equation 4.8. The sub-case $[\![Q]\!]_G \subset$ *smart-eval*(Q, G) does not occur even in the case that $F(Q)$ is a subset of the predicates covered by G_j, as the executor performs triple pattern matching over each partition (Alg. 6.3, line 5) to get exact matches.

(ii) Q is evaluated with Triple Pattern Shipping. For this case, the evaluation of Q is carried out as $TPF(Q, \Omega)$ and Q corresponds to a single triple pattern (which is ensured by the query optimizer). Note that Ω can also be $\emptyset$ when there are no other intermediate results). By hypothesis, $TPF(Q, \Omega)$ does not produce $[\![Q]\!]_G$, which contradicts the definition of the *TPF* server operator.

(iii) Q is evaluated following a Hybrid Shipping. This proof follows from the correctness of the query decomposer and optimizer, the cases (i) and (ii). Without loss of generality, assume that Q is composed of two subqueries Q_1 and Q_2 evaluated using the APIs, and Q_2 is evaluated using the TPF API. From cases (i) or (ii), it follows that *smart-eval*(Q_1, G) is correct and produces the intermediate results Ω. Then, the executor proceeds with the evaluation of Q_2 with intermediate results Ω as $TPF(Q_2, \Omega)$; from case (ii), it follows that *smart-eval*(Q_2, G) is correct. Therefore, we conclude that *smart-eval*(Q, G) is also correct. □

6.5 Experimental Evaluation

We report the performance of `smart-KG+` in comparison to state-of-the-art SPARQL engines over Linked Data Fragments. All datasets, queries, and results, including additional experiments, details on the implementation and configurations used in the experiments are available online[5]. We organize the conducted experiments as follows: First, in Sect. 6.5.1, we present the details of our experimental setup. Next, in Sect. 6.5.2, we present the results of the partition generation. We perform an ablation study to assess the impact of each contribution in Sect. 6.5.3. Subsequently, in Sect. 6.5.4, we

[5] https://github.com/smartkgplus/smartkgplus/tree/master

conduct a performance evaluation of our approach, comparing it to the state of the art. Further, we extend this evaluation in Sect. 4.5.3 to assess the query performance under different query shapes. The resource consumption of our introduced interface is compared to other existing interfaces in Sect. 4.5.4. In Sect. 6.5.7.1, we evaluate typed-family partitioning using multiple datasets. In Sect. 6.6, we summarize the lesson learned from the experimental evaluation.

6.5.1 Experimental Setup

In this section, we present the experimental setup, including the characteristics of the compared systems, the benchmark KGs, the query workloads, the hardware and software configurations, and the evaluation metrics.

6.5.1.1 Compared Systems

- **smart-KG**: We use the Java implementation of `smart-KG` [AFA+20], extending the TPF implementations[6]. HDT indexes and data are stored on the server's disk, with no client-side family caching. This implementation includes:
  - Query Planner: The `smart-KG` client-side query planner generates left-linear plans. This planner relies on the server's partition metadata to determine whether to use the triple pattern or partition shipping. The metadata is transferred to the client-side once before evaluating queries, requiring additional data transfer.
  - Client-side Joining: We implement a joining strategy, following TPF implementation [VSH+16]. The join processing is performed on the client-side based on the client-side query plan.
- **smart-KG+**: We implement both client and server in Java[5], extending `smart-KG`, which includes:
  - Query Planner: We implement server-side query planner to re-order the star-subqueries and triple patterns based cardinality estimations available at the server. Details are presented in Sect. 6.4.1, Alg. 6.1.
  - Client-side and Server-side Joining: We implement a joining strategy, following the brTPF implementation [HA16]. We enable the clients to attach intermediate results to brTPF requests.This enables a distributed join execution between the client and server using the bind join strategy [HKWY97].
- **Triple Pattern Fragments (TPF)**: We use the Java TPF client along with the TPF server [VSH+16].

[6]Linked Data Fragments: `http://linkeddatafragments.org/software/`.

Table 6.1: Characteristics of the evaluated knowledge graphs

RDF Graph G	# triples $\|G\|$	# subjects $\|S_G\|$	# predicates $\|P_G\|$	# objects $\|O_G\|$
WatDiv-10M	10,916,457	521,585	86	1,005,832
WatDiv-100M	108,997,714	5,212,385	86	9,753,266
WatDiv-1B	1,092,155,948	52,120,385	86	92,220,397
DBpedia	837,257,959	113,986,155	60,264	221,623,898

- **SaGe**: We use the Java implementation of both the SaGe server and client. We follow the recommended configurations [MSM19]. Specifically, we configure SaGe to operate with 4 workers, as suggested by the authors.

- **`WiseKG`**: We utilize the `WiseKG` client and server Java implementation, extending the TPF implementations. The `WiseKG` server employs Star Pattern Fragments (SPF) for efficient server-side processing of star-subqueries and uses the family generator from `smart-KG` to manage and store HDT files for family-based partitions. The `WiseKG` client implements a bind join strategy similar to brTPF and SPF, `smart-KG`$^+$ client implementations.

In our experiments, we do not consider SPARQL endpoints, since several previous studies [VSH$^+$16, MSM19, AKMH20] including ours [AFA$^+$20] have already shown that endpoints suffer from scalability problems when increasing the number of clients.

6.5.1.2 Knowledge Graphs

We use various RDF graph datasets including synthetic and real-world datasets. We construct three different dataset sizes including 10M, 100M, and 1B triples from the synthetic dataset Waterloo SPARQL Diversity Benchmark (WatDiv) [AHÖD14]. We design these KG sizes according to the size of open KGs on the LOD Cloud[7], with an average of 183M RDF triples. In addition, we evaluate the compared systems based on a real-world dataset such as DBpedia (v.2015A) [LIJ$^+$15]. We report the characteristics of the evaluated RDF KGs in Table 6.1. In addition, we report statistics on computing family partitioning over other real-world RDF KGs such as WordNet [Fel98], Yago2 [HSBW13], DBLP [Ley02], Freebase [BEP$^+$08]. However, these KGs are not used for assessing the performance of the query engines, as there are no well-known benchmark queries for these datasets.

6.5.1.3 Queries and Workloads

We consider two different query workloads for the synthetic WatDiv datasets:

- A *basic testing* workload denoted as `watdiv-btt` that includes a set of queries extracted from WatDiv basic testing templates[8]. We generate for each client a set of

[7]The Linked Open Data Cloud. https://lod-cloud.net/

[8]`https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml`

20 queries with the following shapes: *linear (L)*, which represents simple path queries; *star (S)*, which includes star queries with at least one instantiated object; *snowflake (F)*, which combines multiple star shapes connected with short paths; and *complex (C)*, which provides challenging queries composed of typically low-selective stars and path queries. Various clients may exhibit query overlap among themselves, but within an individual client, there are no instances of query repetition.

- A *stress testing* workload denoted as `watdiv-sts` comprises a collection of queries sourced from the WatDiv stress-testing suite[9]. Each client workload encompasses a total of 156 non-overlapping queries[10]. These queries were generated using the Waterloo SPARQL Diversity Test Suite (WatDiv), which provides stress testing tools [AHÖD14], allowing us to randomly select queries from the WatDiv stress test query workload in a uniform manner. This workload offers a diverse range of structural and data-driven features [AHÖD14].

In addition, we consider a DBpedia real-world query workload:

- A *real-world testing* workload, named `DBpedia-lsq`, consists of 30 `SELECT` queries per client obtained from the FEASIBLE framework [SMN15]. These queries are derived from real user interactions and were executed on the DBpedia 3.5.1 dataset. FEASIBLE is a benchmark generation framework that receives a query log (LSQ [SAH+15] in our case) and produces a representative set of queries from the log considering both data-driven and structural query features. Since we are interested in highly-demanding queries, we randomly selected 30 BGP queries (out of 259) from FEASIBLE with runtime higher than 1s. We include the results of this workload in our online repository

In order to evaluate the proposed typed-family partitioning, as shown in details in Table 6.2, we derive the following testing workloads from *basic* testing and *stress* testing workloads on Watdiv dataset and a *real-world* testing workload extracted from LSQ query logs based on FEASBLE benchmark framework:

- A *basic typed-family partitioning testing* workload, named as `watdiv-btf`, includes 8 queries derived for each client from `watdiv-btt`. Each query contains at least one star-shaped subquery Q_s with a triple pattern that has a `rdf:type` predicate. We divide this workload into two workloads: the first workload named $\texttt{watdiv-btf}_{bounded}$ includes 4 queries for each client where the object of the triple pattern with `rdf:type` predicate is bounded to a value, the second workload, named $\texttt{watdiv-btf}_{unbounded}$, where the object of the triple pattern with the `rdf:type` predicate is unbounded (i.e. variable).

[9]Waterloo SPARQL Diversity Benchmark. `https://dsg.uwaterloo.ca/watdiv/`
[10]brTPF: `http://olafhartig.de/brTPF-ODBASE2016/`

Table 6.2: Evaluation Workloads Statistics. We provide the total numbers for all the 128 clients

Query Workload	Number of queries	Number of stars	Number of stars with type predicate	Number of stars with *bounded* type predicate
watdiv-sts	19968	35683	6283	3886
watdiv-btt	2560	5248	1152	512
watdiv-btf	1024	1664	1152	512
watdiv-btf$_{bounded}$	512	640	640	512
watdiv-btf$_{unbounded}$	512	1024	512	0
watdiv-stf$_{bounded}$	2944	6144	2944	2944
watdiv-stf$_{unbounded}$	1792	3072	1792	0
watdiv-stf$_{both}$	768	1792	1536	768
DBpedia-lsq	3840	5632	896	768
DBpedia-btt$_{bounded}$	2432	4352	3200	3200
DBpedia-btt$_{unbounded}$	768	768	0	0

- A *stress typed-family partitioning testing* workload derived from `watdiv-sts` named `watdiv-stf`. We include a set of queries that contain at least one star-shaped subquery Q_s with the `rdf:type` predicate. We divide the obtained queries into three different workloads. The first workload, named `watdiv-stf`$_{bounded}$, includes 23 queries for each client where the object of the triple pattern with type predicate is bounded. The second workload, `watdiv-stf`$_{unbounded}$, includes 14 queries for each client where the object of the triple pattern with `rdf:type` predicate is a variable. The third workload, `watdiv-stf`$_{both}$, contains 6 queries per client, where one star-subquery involves a bounded object in the triple pattern with the type predicate, while another star-subquery includes an unbounded object in the triple pattern with the type predicate.

- A *real-world typed-family partitioning testing* workload. We extract 25 real-users SELECT queries for each client from FEASIBLE [SMN15] benchmark on the DBpedia 3.5.1 dataset. Note that we make sure that the queries are compatible with our DBpedia dataset version, v.2015A. We selected queries that contain at least one-star pattern with at least one triple pattern with the `rdf:type` predicate. We divide the selected queries into two workloads. The first workload, named as `DBpedia-btt`$_{bounded}$, consists of 19 queries with at least one-star query with a bounded type predicate. The second workload, named as `DBpedia-btt`$_{unbounded}$, consists of 6 queries for each client with at least one-star query with an unbounded `rdf:type` predicate.

6.5.1.4 Hardware Setup

- **Client specifications:** We design experiments with an increasing number of clients following eight configurations with 2^i clients ($0 \leq i \leq 7$) issuing concurrent queries to the server. Each client executes one query at a time, i.e., the server receives at most 128 queries simultaneously. We ran all eight configurations 1, 2, 4, 8, 16, 32, 64, and 128 clients concurrently on a virtual machine with 128 vCPU cores of 2.5GHz, 64KB

L1 cache, 512KB L2 cache, 8192KB L3 cache, and 2TB main memory. To ensure equal resource allocation among the clients, we bound each client (for the compared systems) to a single vCPU core and 15GB of main memory.

- **Server specifications:** The compared systems servers run on a virtual machine (VM) hosted on a machine with 32 3GHz vCPU cores, 64KB L1 cache, 4096KB L2 cache, 16384KB L3 cache, and 128GB main memory. To ensure that enough resources are left for the VM, it was made sure that the hypervisor was not over-committing resources. Furthermore, KVM processor affinity was configured so that each VM would be only using a set of explicitly defined CPU cores, ensuring that other VMs running on the hyper-visor are not using the resources of the VM running the SPARQL servers.
- **Network configuration:** While clients and servers are connected over a 1 GBit Ethernet network, we bound the network speed of each client to 20MBit/sec to emulate a practical bandwidth offered by internet service providers.

6.5.1.5 Evaluation Metrics

- **Throughput**: Number of workload queries completed per minute.
- **Timeouts (TO)**: Number of workload queries that exceed the timeout. We set timeout thresholds of 5 and 30 minutes for WatDiv and DBpedia queries, respectively.
- **Workload Completion Time**: Total elapsed time required by a client to execute an entire query workload.
- **Query Execution Time (ET)**: Average elapsed time to execute a single query in a query workload.
- **First Result of a Query**: Elapsed time to retrieve the first result of a query in a query workload.
- **Server CPU load**: The average percentage of server CPU used during the execution of a query workload.
- **Number of Requests (Req)**: Total number of requests received by the server from a client.
- **Number of Transferred Bytes (DT)**: Total number of bytes transferred on the network between the server and clients.

6.5.2 Creation of Family-based Partitions

Table 6.3 presents the thresholds used for creating the family-based partitions for each KG G. Note that the configuration $(\alpha_s, \alpha_t, \tau_{p_{low}}, \tau_{p_{high}}, \tau_{class_{low}}, \tau_{class_{high}}) = (0, |G|, 0, 1, 0, 1)$ corresponds to full materialization of all families. For the smallest dataset WatDiv-10M, we tested this configuration. Then, we assess the impact of the smart-KG$^+$ family

Table 6.3: Family-based Partitions Parameter Settings

RDF Graph G	α_s	α_t	$\tau_{p_{low}}$	$\tau_{p_{high}}$	$\tau_{class_{low}}$	$\tau_{class_{high}}$	$\lvert P'_G \rvert$	$\lvert P'_{core} \rvert$	$\lvert F'_{core} \rvert$	$\lvert \mathcal{G}_{serv} \rvert$	C. Time (h)
WatDiv-10M	0	$\lvert G \rvert$	0	1	0	1	85	85	13,002	38,400	2
WatDiv-10M	0	$0.05\lvert G \rvert$	0.01/100	1	0	1	59	59	10,106	21,210	1
WatDiv-100M	0	$0.05\lvert G \rvert$	0.01/100	1	0	1	59	59	22,855	37,392	7
WatDiv-1B	0	$0.05\lvert G \rvert$	0.01/100	1	0	1	59	59	39,046	52,885	12
DBpedia	0.01/100	$0.05\lvert G \rvert$	0.01/100	0.1/100	0.01/100	0.1/100	218	84	35	29,965	23

pruning strategies with the following set up. We empirically set $\alpha_t = 0.05|G|$ for all datasets, to avoid large families containing more than 5% of the triples in G. Then, we use $\alpha_s = 0$ for WatDiv to allow all families (even small ones), but $\alpha_s = 0.01/100$ for DBpedia to create families where the predicates appear in at least 0.01 of the subjects in the graph. Likewise, we fixed $\tau_{p_{low}} = 0.01/100$ for all G, while we set $\tau_{p_{high}} = 0.1/100$ for DBpedia, as we empirically tested that the resultant predicate set filters out both infrequent and heavy hitters. We refer to [AFA+20] for a study on DBpedia on the number of families with different values of our parameters. Lastly, for typed families, we tested the parameters $\tau_{class_{low}} = 0.01/100$ and $\tau_{class_{high}} = 0.01/100$ for DBpedia, applied to 376 classes selected based on an empirical test that the resultant class set filters out heavy hitter classes as well as infrequent classes.

For each graph G, Table 6.1 also shows the number of restricted and core predicates ($|P'_G|$, $|P'_{core}|$), core families, $|F'_{core}|$, and the materialized partitions after grouping/pruning, $|\mathcal{G}_{serv}|$, as well as the total computation time (including family computation, pruning, and partition generation). Table 6.1 also shows that $|F'_{core}|$, $|\mathcal{G}_{serv}|$, and the computation time are sub-linearly increasing with the graph sizes. In WatDiv, $F'_{core} = F'(G)$, whereas in DBpedia, the initial number of P'_G-restricted[11] families $|F'(G)|$ is >600K: the family pruning strategy allows `smart-KG+` to identify $|F'_{core}| = 35$ core families, which are merged into ∼30K materialized partitions. We provide an analysis of the impact/coverage of different parameter values for the case of DBpedia in our online repository[5]. Lastly, we present the results of family creation in further real-world KGs, i.e., Yago2, WordNet, and DBLP. In Table 6.4, we present additional real-world Knowledge Graphs (KGs) partitioned using family-based techniques. Freebase and Yago2 follow the parametrization of DBpedia due to their similar characteristics (see Table 6.3). DBLP and WordNet use the same setup as WatDiv due to their comparable characteristics.

6.5.3 Ablation Study: Assessing the Impact of the `smart-KG+` Components

In this section, we conduct an ablation study to evaluate the performance of each individual contribution made to `smart-KG+`. The goal is to gain insights into the

[11] The 218 restricted DBpedia predicates cover over 40% of the predicates occurring in highly-demanding BGPs (>1s of execution time) in the real-world LSQ query log [SAH+15].

Table 6.4: Characteristics of additional real-world knowledge graphs

RDF Graph G	# triples $\|G\|$	# subjects $\|S_G\|$	# predicates $\|P_G\|$	# objects $\|O_G\|$	$\|P'_G\|$	$\|P'_{core}\|$	$\|F'_{core}\|$	$\|\mathcal{G}_{serv}\|$	C.Time (h)
Freebase	2,067,068,155	102,001,451	770,415	438,832,462	530	171	479	11979	18
Yago2	158,991,568	67,813,972	104	22,354,760	35	19	65	638	5
DBLP	88,150,324	5,125,936	27	36,413,780	27	27	270	990	3
WordNet	5,558,748	647,215	64	2,483,030	64	64	777	1156	0.5

Table 6.5: An ablation study to assess the performance of each individual contribution over `watdiv10M` using `watdiv-btt` workload. (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in ms, TO: Timeouts). GM-T = Total Geometric mean for all query classes

	smartKG (brTPF +NP)			smartKG (TPF +NP)			smartKG (TPF +OP)		
Query	Req	DT	ET	Req	DT	ET	Req	DT	ET
L1	**4**	**0.54**	**206**	28	1.1	333	60	0.54	218
L2	3	0.34	175	**3**	0.34	188	**2**	0.34	**51**
L3	**2**	0.5	579	**2**	0.5	566	**2**	0.5	**28**
L4	**2**	0.5	188	**2**	0.5	169	**2**	**0.48**	**69**
L5	3	0.16	192	3	0.16	221	**2**	0.16	**52**
S1	3	0.13	221	3	0.13	206	**2**	**0.12**	**66**
S2	**2**	0.22	191	**2**	0.22	204	**2**	0.22	**117**
S3	**2**	0.59	209	**2**	0.59	181	**2**	**0.55**	**61**
S4	**10**	**0.48**	**226**	16	0.66	575	216	0.72	1476
S5	**2**	0.42	**161**	**2**	0.42	163	**2**	**0.39**	3863
S6	**2**	**0.01**	170	**2**	**0.01**	**141**	699	3.8	7508
S7	2	0.003	**221**	2	0.003	224	2	0.003	**52**
F1	**4**	**0.9**	240	**4**	0.98	**228**	15	2.95	452
F2	3	**0.9**	184	3	0.91	**179**	**2**	1.2	361
F3	**5**	1.5	311	1503	**0.16**	**268**	2541	1.4	298
F4	**5**	**0.8**	**217**	2029	30.56	31858	24000	0.81	76073
F5	5	5.8	**247**	5	5.8	282	**3**	5.8	2478
C1	**4**	**6.9**	**372**	**4**	**6.9**	384	**6**	7.4	683
C2	39218	52.1	300071	3181	52.1	**171848**	102441	3.1	208721
C3	**2**	0.8	**21635**	**2**	0.81	22200	**2**	0.8	28316
GM-T	**4.9**	**0.5**	**407.7**	8.76	0.57	539.03	18.82	0.64	605.12

significance of each change introduced w.r.t. the earlier version. For this purpose, we developed three configurations of the interface:

- **TPF+OP:** This configuration represents the early version of `smart-KG`, combining TPF with client-side query planning (OP).
- **TPF+NP:** This configuration is a variant of our smartKG interface that allows us to observe the impact of the new server-side query planning (NP) while using TPF.
- **brTPF+NP:** This configuration represents our proposed solution `smart-KG`$^+$, which combines brTPF with server-side query planning (NP).

To assess the performance of these configurations, we conduct a performance evaluation using `watdiv-btt` and `watdiv-sts` on `watdiv10M`; results are presented in Table 6.5 and Table 6.6.

Table 6.6: An ablation study to assess the performance of each individual contribution over watdiv10M using watdiv-sts workload. (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in ms, TO: Timeouts). GM-T = Total Geometric mean for all query classes

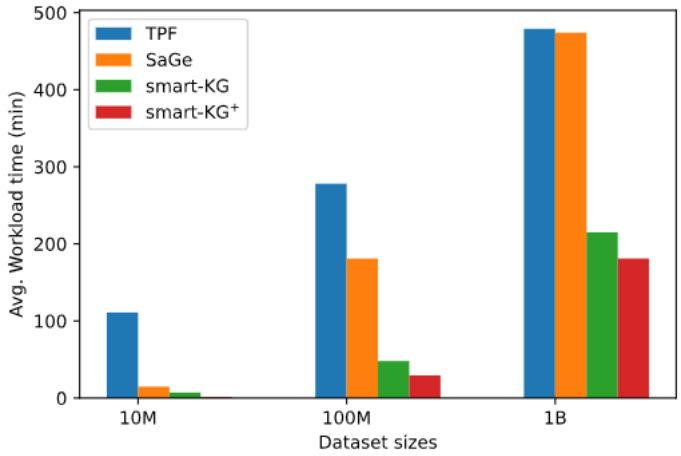
Workload	smartKG (brTPF + NP)				smartKG (TPF + NP)				smartKG (TPF + OP)			
	Req	DT	ET	TO	Req	DT	ET	TO	Req	DT	ET	TO
watdiv-sts	**554**	**203.41**	**24.405**	6	24722	546.26	382.236	**0**	3768	387.953458	55.876	**0**

In Table 6.5, we observe that the smart-KG outperforms other approaches in handling simple linear queries (L1 - L5) and highly selective star queries (S1 - S3). This performance superiority is attributed to the comparatively lower average execution time of 82 milliseconds for these queries, while the query planning process in our proposed solution, smart-KG+, requires an average of 70 milliseconds. However, it is essential to consider that client-side query planning requires an initial data transfer of 1.75MB on average, comprising metadata that represents the family partitioning of the queried knowledge graph. This metadata is crucial for identifying the required partition for each query. Shipping the metadata file demands an average of 700 milliseconds, but it can be cached locally and subsequently utilized for multiple queries. It is important to note that the performance results presented for smart-KG presume that the metadata file is already stored on the client-side.

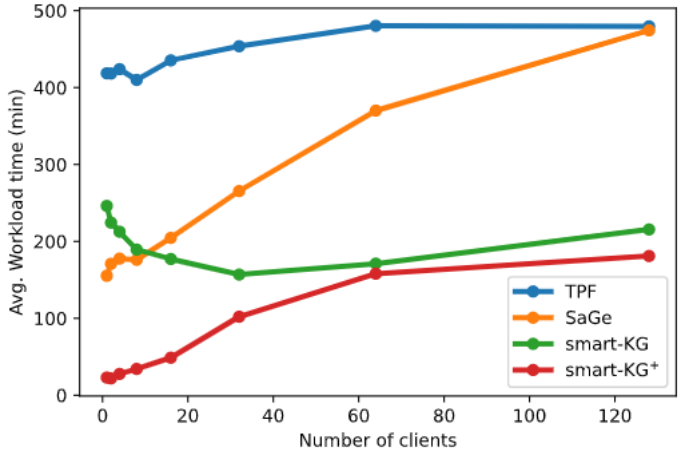
Table 6.5 also shows a significant improvement in performance, with up to a 50% reduction in execution time observed, for both systems reliant on the server-side query planner for F queries comprising 2-3 stars per query. This improvement can be attributed to our server-side query planner's utilization of star reordering based on characteristic sets, which offers a better reordering compared to the one achieved by smart-KG in the case of snowflake queries.

In the case of C queries, C1 demonstrates performance enhancement through the adoption of the star-reordering technique provided by the query planner of smart-KG+. However, for C2, brTPF+NP exhibits slightly lower performance compared to other systems. This is attributed to the query execution strategy of smart-KG+, which always pushes intermediate results to brTPF, instead of joining the intermediate results entirely at the client-side. This lead to unnecessary requests in C2, resulting in a longer runtime. Still, brTPF+NP provides the best total geometric mean for the number of requests, data transfer, and execution time compared to the other two versions.

In Table 6.6, we present the performance analysis of three different configurations applied to the stress workload watdiv-sts. The results demonstrate a significant improvement in the performance of smart-KG+ (brTPF +NP) when compared to the other two versions. We note that the smart-KG+ (brTPF +NP) version experienced 6 timeouts, whereas the remaining versions did not encounter any timeouts. These timeouts results from the following reasons. First, low selective queries may time out due to the strategy of attaching large intermediate results back to the server. Second, the process of attaching the intermediate results to the brTPF request incurs higher costs compared to a regular TPF request. This increased cost further affects the overall performance and contributes



(a) Average Workload Completion Time of 128 concurrent clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets on `watdiv-sts` workload

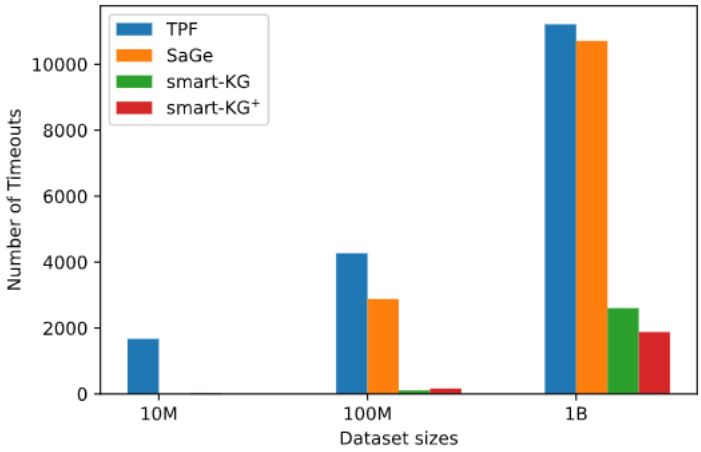


(b) Average Workload Completion Time for increasing numbers of clients over `watdiv1B` on `watdiv-sts` workload

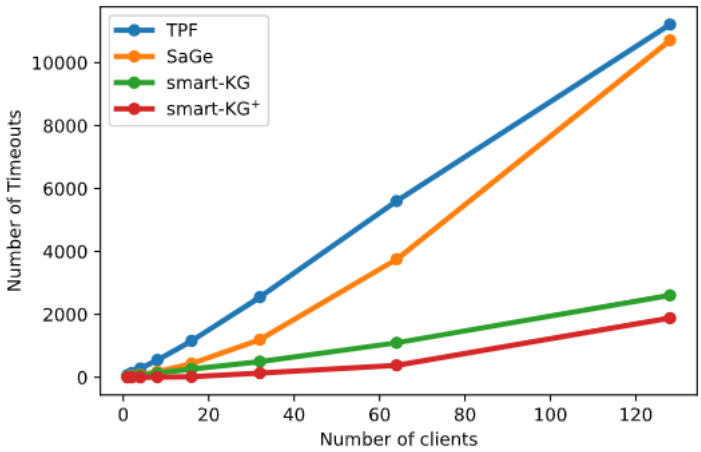
Figure 6.5: Workload completion time (lower is better)

to the occurrence of timeouts. Third, a mismatch in query planning can potentially lead to longer execution times. This was observed in two queries in `TPF+NP`, which took more than a minute to execute, as well as in the case of `C2`.

To conclude, the new query planner finds better query plans but with the cost of a server request. In addition, our strategy to query brTPF achieves better performance in queries that require shipping a small number of intermediate result, while querying TPF achieves better performance for queries that require shipping many intermediate results.



(a) Number of timeouts of 128 concurrent clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets on `watdiv-sts` workload



(b) Number of timeouts for increasing numbers of clients over `watdiv1B` on `watdiv-sts` workload

Figure 6.6: Number of timeouts (lower is better)

6.5.4 System Performance Evaluation

In this section, we evaluate the performance on increasing number of concurrent clients (up to 128 clients) on three different graph sizes including `watdiv10M`, `watdiv100M`, and `watdiv1B` using `watdiv-sts` workload. For these experiments, and the ones presented in Section 6.5.5 and Sect. 6.5.6, `smart-KG+` does not use the typed-partitions. This allows for measuring the impact of the new planning and pipelined join strategies implemented, and comparing them to the previous techniques implemented in `smart-KG`.

Workload Completion Time Analysis. Fig. 6.5 shows the average workload completion time results of executing the `watdiv-sts` workload including the queries that have

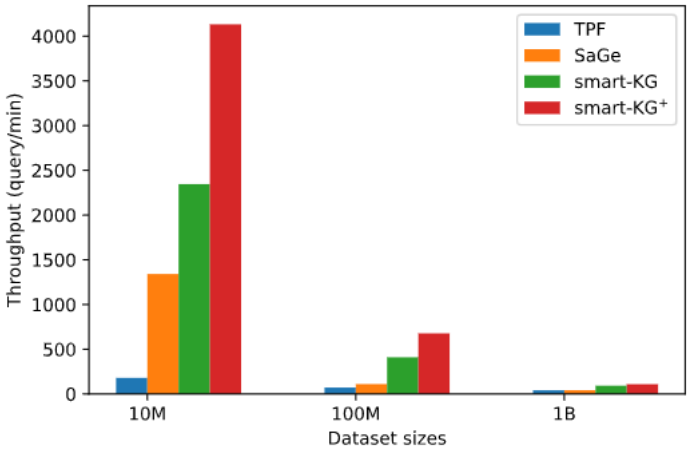


Figure 6.7: Query throughput of 128 concurrent clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets on `watdiv-sts` workload

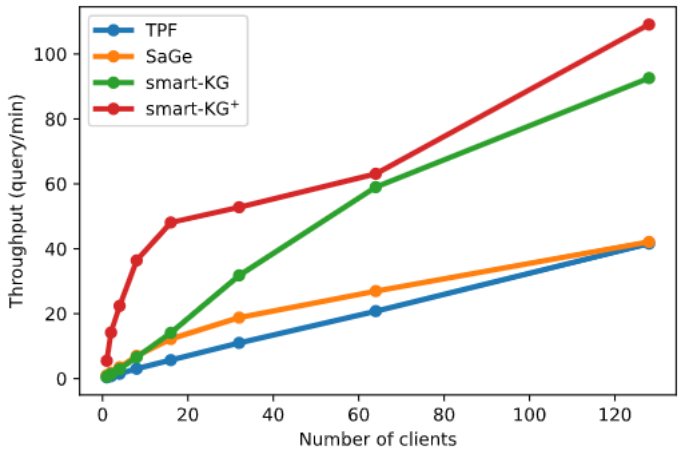


Figure 6.8: Query throughput for increasing numbers of clients over `watdiv1B` on `watdiv-sts` workload

Figure 6.9: Throughput (higher is better)

timed out. Fig. 6.5a shows the scenario of increasing KG size with the highest number of concurrent clients (128 clients) on using `watdiv-sts`. `smart-KG`$^+$ is up to 7, 2, and 1.3 times faster than `smart-KG` on `watdiv10M`, `watdiv100M`, and `watdiv1B` datasets, respectively. This improvement in performance is due to performing query planning on the server-side, which results in fewer intermediate results transferred over the network.

As shown in Fig. 6.5b, `smart-KG`$^+$ provides a significant performance improvement compared to all systems; `smart-KG`$^+$ has an outstanding performance over `watdiv1B`

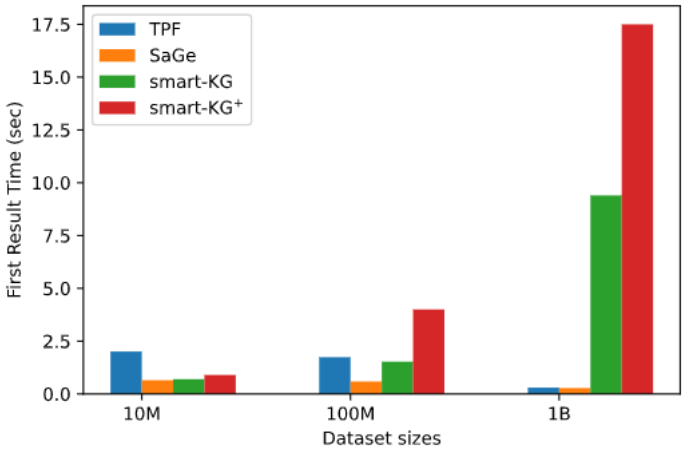


Figure 6.10: Average first result time for 128 clients over increasing sizes datasets on `watdiv-sts` workload

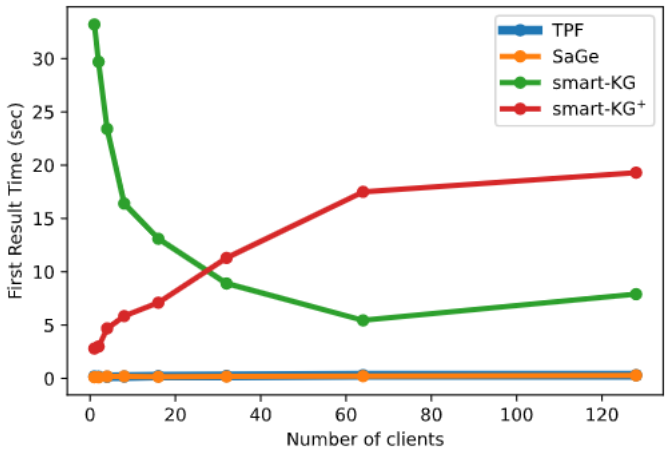


Figure 6.11: Average first result time for increasing number of clients over `watdiv1B` dataset on `watdiv-sts` workload

Figure 6.12: Average first result time (lower is better)

dataset from 1 up to 32 clients compared to `smart-KG` since `smart-KG+` utilizes brTPF which significantly reduces the number of HTTP requests. Note that `smart-KG` performance slightly improves with an increasing number of concurrent clients since TPF request sent by `smart-KG` clients has a higher potential for a cache hit than brTPF request sent by `smart-KG+` client since the HTTP caching is designed to serve the identical requests to earlier ones without the need to access the server to recompute the response over again.

Overall, `smart-KG+` provides a faster workload completion time on using `watdiv-sts` than TPF and SaGe in all experiment setups from 1 up to 128 clients over `watdiv1B`

dataset. smart-KG$^+$ is up to 18 and 7 times faster in the case of 1 client workload, and 3 and 2.6 times with 128 concurrent client workloads than TPF and SaGe, respectively. For less than 16 concurrent clients, SaGe provides a slightly faster workload completion time than smart-KG. From this point forth, SaGe suffers from performance degradation due to the excessive waiting queue time of the round-robin policy.

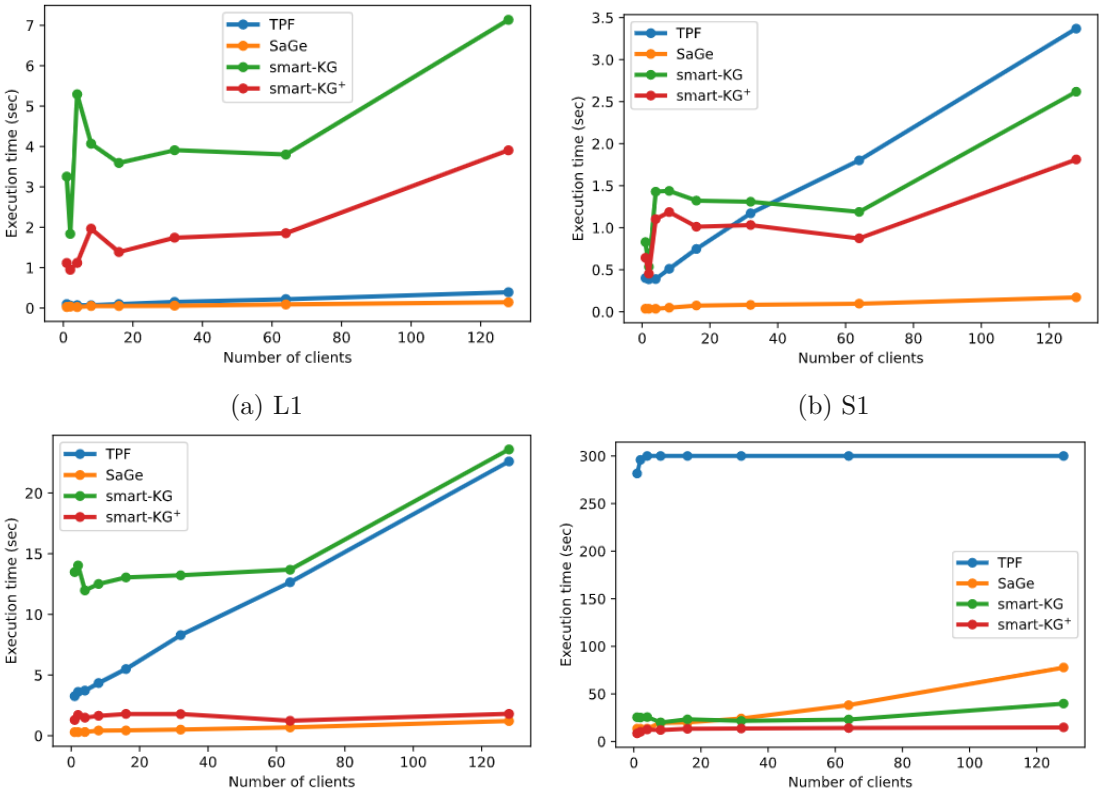
Timeout Analysis. Fig. 6.6a illustrates that smart-KG and smart-KG$^+$ produces relatively low timeouts compared to the state-of-the-art system TPF and SaGe. That is, with 128 concurrent clients, smart-KG$^+$ and smart-KG have approximately a percentage of 9% and 13% of watdiv-sts workload queries timed out over watdiv1B dataset. In contrast, as shown in Fig. 6.6b, on watdiv1B, the percentage of timeouts drastically increases for TPF with an increasing number of clients, from 44% in 1-client workload to 56% with 128 clients. Similarly, the percentage of timeouts rises rapidly from 10% with 1 client up to 54% with 128 concurrent clients.

As expected, the number of timeouts of TPF and SaGe has excessively increased with the size of the RDF KG size. On watdiv10M, SaGe produces no timeouts while TPF has a percentage of 10% timeouts. In turn, SaGe timeouts increase substantially with increasing KG sizes, with a trend similar to TPF over watdiv100M and watdiv1B.

Throughput Analysis. We consider throughput as a metric to explore the performance of the systems under high load, i.e., an increasing number of concurrent clients and the sizes of the KGs. We measure the throughput as the total number of queries executed per minute from all concurrent clients. Note that we consider the queries that have terminated successfully and provided complete results within the predetermined timeout limit.

Fig. 6.7 shows that smart-KG$^+$ achieves an higher throughput values than all the compared systems over different KG sizes reaching 4132, 678, 109 query/min over watdiv10M, watdiv100M and watdiv1B, respectively. In Fig. 6.8, we observe two main findings. First, smart-KG$^+$ scales better than all the compared systems since it has a higher query throughput with an increasing number of clients. Second, all compared systems are able to achieve higher throughput with an increasing number of clients, which shows that the systems can scale well but at a different rate.

First Result of a Query Analysis. Fig. 6.10 shows that SaGe provides the best response time to all systems over different sizes of KGs. This is not surprising since SaGe is, in principle, a SPARQL endpoint that adopts a Web preemption technique to avoid the convoy effect phenomenon caused by the long-running queries. smart-KG$^+$ provides a comparable query response time to smart-KG and SaGe and even slightly better than TPF on watdiv10M dataset. However, as the KG size increases, the average response time also increases for watdiv100M and watdiv1B. This can be attributed to two factors. First, the larger sizes of the shipped KG partitions result in longer download times. Second, smart-KG$^+$ relies on brTPF for handling single triple pattern fragments, unlike the previous version smart-KG which used TPF. While brTPF potentially requires fewer requests compared to TPF, it introduces additional time for attaching and parsing



(a) L1 (b) S1

(c) F1 (d) C1

Figure 6.13: Avg. execution time per client on WatDiv-100M, for the first query of each category L, S, F, and C

solution mappings. Moreover, brTPF utilizes the binding join strategy to distribute the workload between clients and the server. Consequently, with an increasing number of clients, brTPF puts more load on the server compared to TPF, resulting in slightly slower query response times as shown in Fig. 6.10.

In Fig. 6.11, we observe that TPF and SaGe have an almost constant curve (i.e. negligible response time increase) with an increasing number of clients. In turn, `smart-KG+` has on average a longer response time between 2 seconds on 1 client workload and 17 seconds on 128 clients workload. As a final noteworthy observation regarding the response time metric, `smart-KG` response time is actually decreasing with an increasing number of concurrent clients, as we discussed earlier, the likelihood of a cache hit for identical TPF requests from different query execution is higher than brTPF requests. In other words, with an increasing number of clients, TPF requests issued by `smart-KG` clients are more frequently answered from an HTTP cache that acts as a proxy server than brTPF requests issued by `smart-KG+`. This is consistent with the results reported by Hartig and Buil-Aranda [HA16].

Table 6.7: Avg. execution time per client (in sec.) for 128 clients over `watdiv100M` for the `watdiv-btt` workload. GM=Geometric Mean per query class. GM-T = Total Geometric mean for all query classes.

Query	L1	L2	L3	L4	L5	GM-L
TPF	**0.39**	268.7	0.16	35.9	117.18	9.3
SaGe	0.141	11.27	**0.26**	6.86	7.47	**1.84**
smart-KG	7.13	20.66	0.88	2.89	0.94	3.23
smart-KG$^+$	3.90	**5.9**	0.92	**1.99**	**0.875**	2.05

(a) L Queries

Query	F1	F2	F3	F4	F5	GM-F
TPF	22.60	44.35	41.8	50.27	2.21	21.5
SaGe	**1.21**	**0.93**	**1.67**	**2.33**	**0.37**	**1.1**
smart-KG	23.58	7.19	28.19	7.17	7.83	12.18
smart-KG$^+$	1.8	1.832	3.34	2.75	2.27	2.39

(b) F Queries

Query	S1	S2	S3	S4	S5	S6	S7	GM-S
TPF	3.36	59.2	38.063	36.91	92.39	9.58	0.034	9.75
SaGe	**0.17**	2.79	10.85	**2.9**	5.70	**0.77**	**0.09**	**1.28**
smart-KG	2.61	0.99	0.91	43.02	3.62	67.41	0.97	4.22
smart-KG$^+$	1.81	**0.83**	**0.402**	23.02	**2.8**	3.6	0.43	1.79

(c) S Queries

Query	C1	C2	C3	GM-C	GM-T
TPF	300.0	300.0	510.37	358.13	130.2
SaGe	77.74	74.18	480.10	140.41	55.12
smart-KG	39.85	300.0	363.35	163.16	68.19
smart-KG$^+$	14.85	**48.61**	**260.11**	**57.26**	**29.37**

(d) C Queries

6.5.5 Performance evaluation on different query shapes

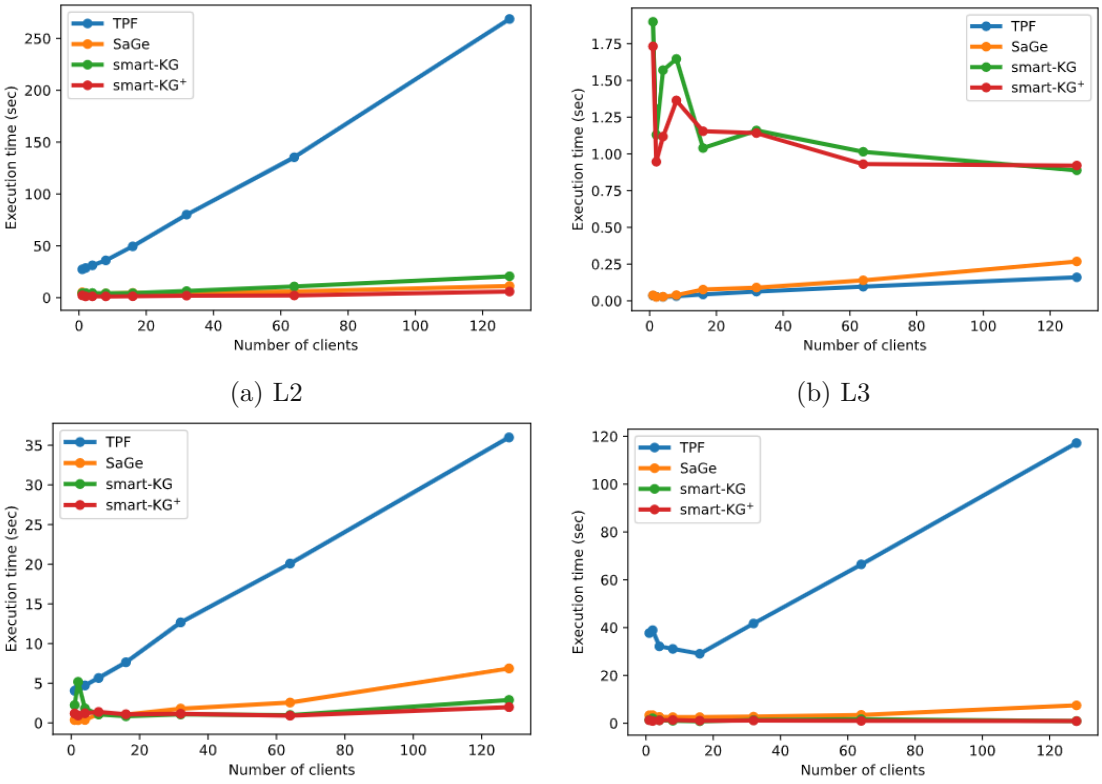
In this section, we investigate the query performance of the compared systems on four different query shapes previously introduced by the *WatDiv Basic Testing* [AHÖD14]. In the following, we provide an overview of the trend of the average execution time of each category in Fig. 6.13.

In general, SaGe has an outstanding performance for all query shapes. This behavior can be explained by the size of the workload; the `watdiv-btt` workload includes only 20 queries per client inducing a low query arrival rate to the SaGe server. In contrast, TPF is significantly worse than most of the compared systems except in simple queries due to shipping large intermediate results and a high number of requests. In turn, `smart-KG` provides a relatively slow performance in L, S, and F queries since it ships partitions with unnecessary intermediate results for such selective queries. Yet, `smart-KG`$^+$ provides an efficient query performance in F and C queries. Interestingly, although `smart-KG`$^+$ still has to ship the same partitions as `smart-KG`, `smart-KG`$^+$ provides better performance in most query shapes thanks to the more accurate query planner. As expected, the performance of `smart-KG`$^+$ enhances gradually from simple L queries reaching its best performance in complex C queries.

Table 6.7 summarises the average execution time for all different query shapes over watdiv100M with 128 clients. For the L-workload, smart-KG$^+$ and SaGe offer comparable performance in L, with a better geometric in the simplest queries and highly selective queries with a small diameter. For the F queries, SaGe provides the best performance in the F-workload compared to all systems. In turn, smart-KG$^+$ outperforms all the compared systems in C queries, especially in C2, where both smart-KG and TPF timeouts due to the large intermediate results. Finally, smart-KG$^+$ achieves the smallest total geometric mean among all the compared systems and all query shapes (cf. Table 6.7, GM-T column).

Fig. 6.14 shows the performance in the simplest L-queries of the different systems on WatDiv-100M. Similar to our previous results, smart-KG$^+$ reports a stable query execution time, which ranges between 1-5 seconds. smart-KG$^+$ performs better than the original smart-KG due to the asynchronous pipeline of iterators executing first the most selective iterator. As expected, SaGe provides excellent performance in L queries (i.e. simple queries), with the best performance in the L3 query with an average execution time of less than 1 second. The main reason is that SaGe server in the case of L queries acts as a SPARQL endpoint since it requires a single request to process L query. Finally, TPF is the slowest approach in L2, L4, L5 queries, while it excels in L1 and L3 up to 40 clients since the queries are very selective and do not require pagination.

Fig. 6.15 shows the query execution time of S-queries. smart-KG$^+$ provides a more efficient performance than smart-KG, since in this case, smart-KG$^+$ server query planner generates far more accurate triple pattern ordering than smart-KG, relying on pre-computed cardinality estimations (i.e. characteristics sets) stored on the server-side. SaGe maintains a solid performance in S-queries (very selective) requiring less time on the server. As shown in figure 6.16, SaGe provides the best execution time for F queries (i.e snowflake queries). smart-KG has on average a slow query execution time in F queries (i.e snowflake queries) since snowflake queries require a join operation between the shipped stars which are typically connected with a non-selective single triple pattern evaluated by a high number of TPF requests. However, smart-KG$^+$ significantly outperforms TPF and smart-KG thanks to the accurate server-side query planning. Finally, Fig. 6.17 shows the overall execution times for the C-queries workload on (WatDiv-100M, 80 clients, 5min timeout). TPF is again the slowest solution, while smart-KG$^+$ significantly outperforms all the compared systems. For instance, smart-KG timeouts at C2 since the query includes 3 stars and 3 single triple patterns with high cardinalities causing a tremendous number of TPF requests. smart-KG$^+$ avoids the large intermediate results by better subqueries reordering. For C3 (unbounded star query), smart-KG and smart-KG$^+$ provide the best performance since they are optimized for star queries. In contrast, SaGe suffers from additional delays in case of complex queries to maintain the fair resources allocation policy.



(a) L2 (b) L3

(c) L4 (d) L5

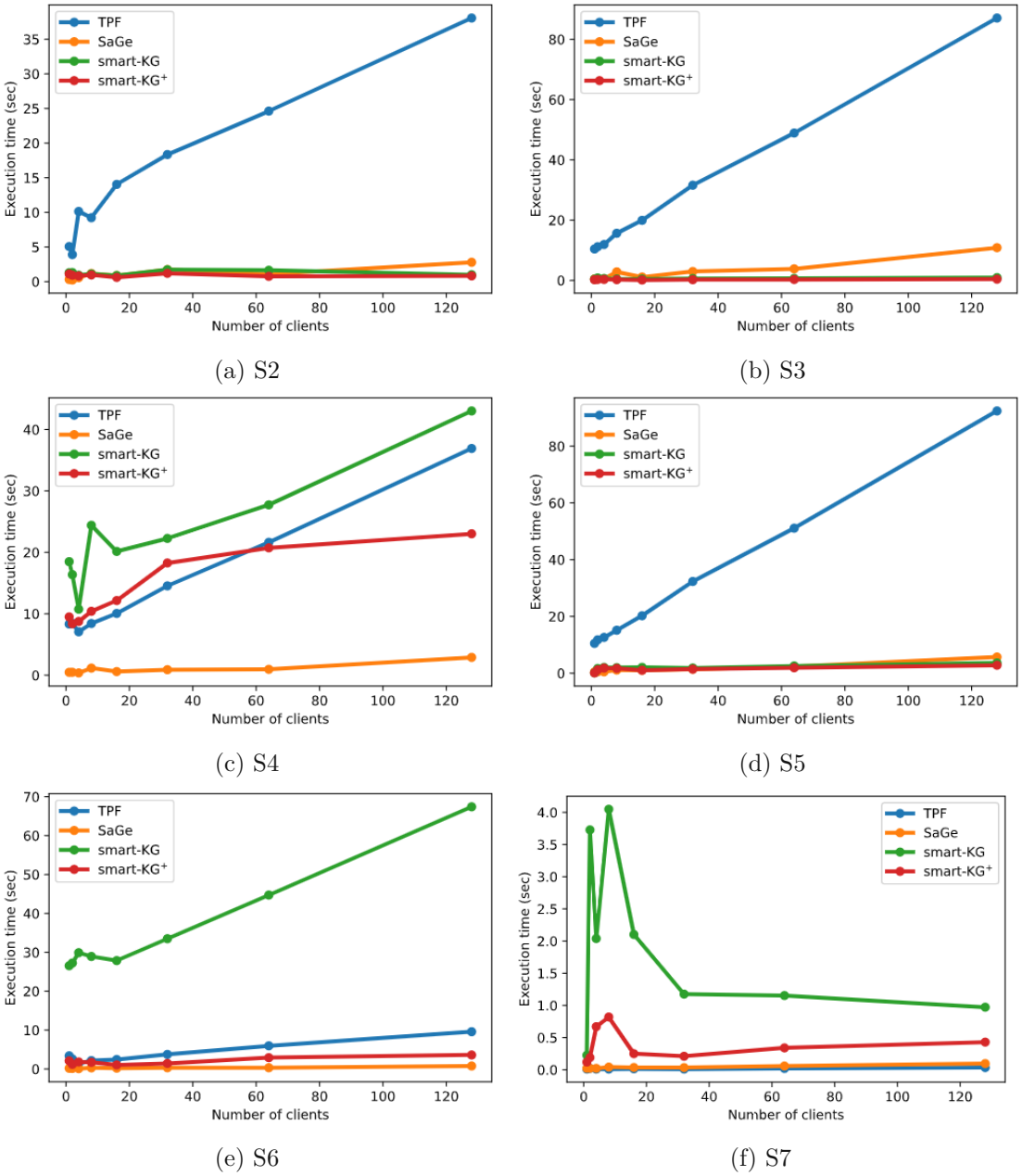
Figure 6.14: Avg. execution time per client on the standard WatDiv-100M, for simplest L queries

6.5.6 Resource Consumption

Network Load. We report two main metrics to describe the network load: the total number of requests received by the server and the number of bytes transferred on the network between clients and the server. The results reported in the following do not account for queries that timed out.

Fig. 6.18a shows the distribution of the number of transferred bytes on increasing KG sizes with 128 concurrent clients. SaGe transfers the least number of bytes over the network compared to all state-of-the-art systems since SaGe acts as a full SPARQL endpoint with a Web preemption as an additional feature to prevent query execution starvation with no intermediate results. SaGe only consumes a small extra data transfer overhead to send query plans of a long running-query in order to enable the clients to resume query execution afterwards. In contrast, TPF incurs the highest data transfer cost due to the enormous amount of shipped intermediate results leading to low query execution performance as already shown in Fig. 6.5 and Fig. 6.6.

`smart-KG+` requires less data transfer than `smart-KG`. This is expected for two main



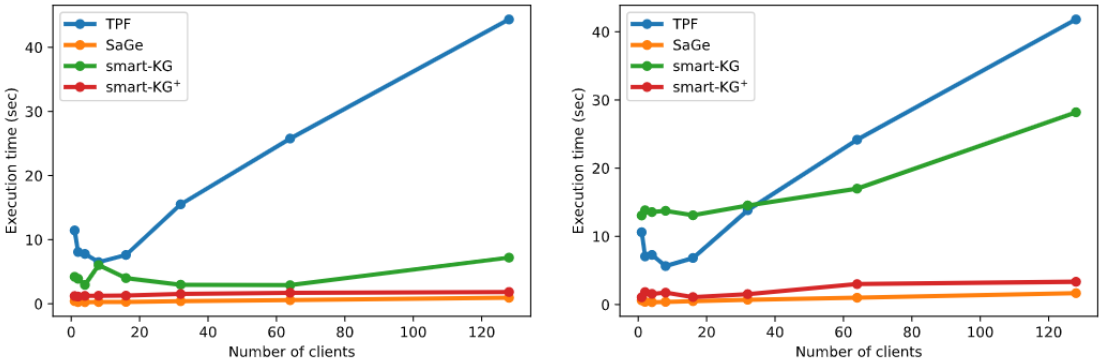
(a) S2 (b) S3

(c) S4 (d) S5

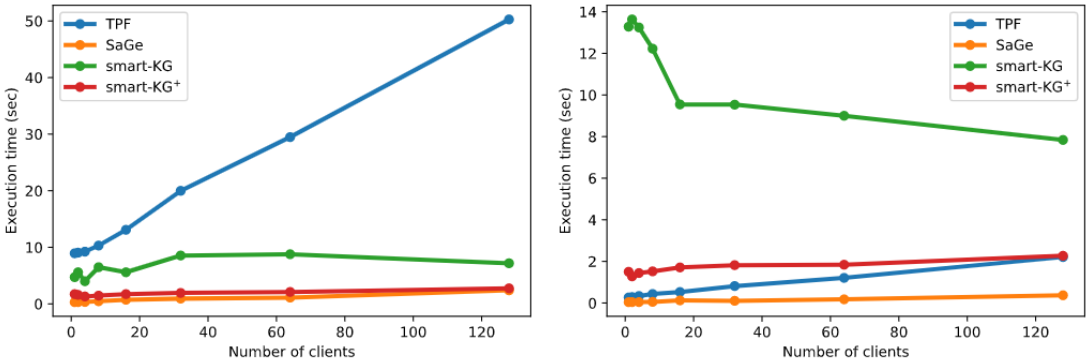
(e) S6 (f) S7

Figure 6.15: Avg. execution time per client on the standard WatDiv-100M, for Star S queries

reasons. First, `smart-KG+` utilizes a star pattern reordering based on cardinality estimation which eventually reduces the intermediate results transferred on the network. Second, `smart-KG+` employs brTPF to handle single non-star triple patterns which reduces the data transfer compared to TPF. To be precise, `smart-KG+` requires to transfer on average 8.1MB and 86.8MB per query over `watdiv100M` and `watdiv1B`.

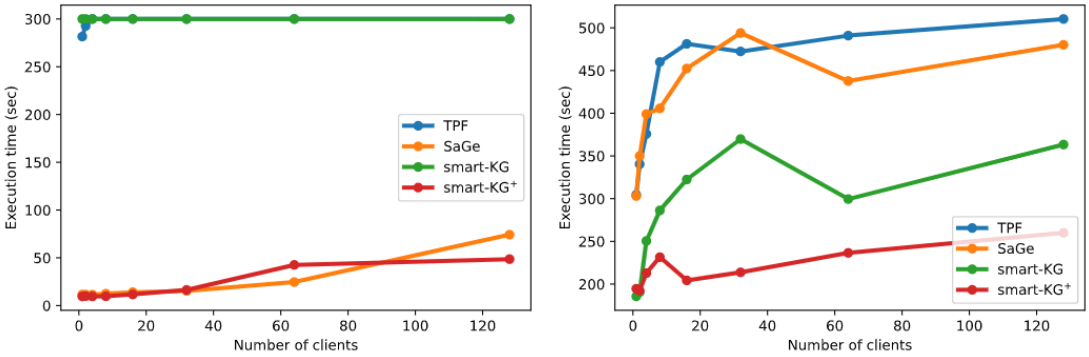


(a) F2 (b) F3



(c) F4 (d) F5

Figure 6.16: Avg. execution time per client on the standard WatDiv-100M, for Snowflake F queries



(a) C2 (b) C3

Figure 6.17: Avg. execution time per client on the standard WatDiv-100M, for Complex C queries

As expected, `smart-KG+` transfers more data over the network than SaGe, but up to 87% and 40% less data than TPF and `smart-KG` per query over `watdiv100M` dataset.

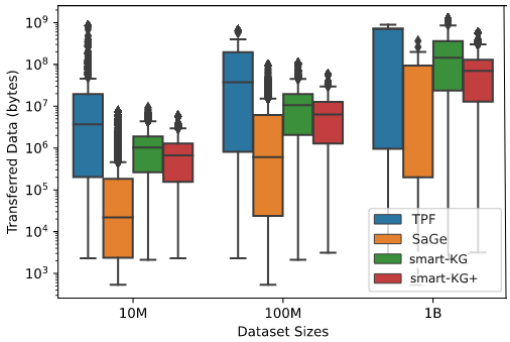
Unlike SaGe, smart-KG$^+$ can leverage the transferred partitions by reusing them in future queries.

As shown in Fig. 6.18b, smart-KG$^+$ significantly reduces the number of requests in comparison to all of the compared systems. smart-KG$^+$ requires on average 8, 17 and 178 requests over watdiv10M, watdiv100M, and watdiv1B. In contrast, TPF incurs an enormous number of requests, reaching more than $10 - 30K$ requests per query (on average) over the different WatDiv datasets. For SaGe, the number of requests considerably increases as a consequence of the scheduling mechanism to allocate server resources among the workload queries. In both versions of smart-KG, the implementation of a caching mechanism would potentially yield a substantial performance improvement. Two strategies can be employed: server-side caching of popular families in-memory and client-side caching, where families are stored locally upon shipment, enabling their reuse for subsequent queries involving the same families. Caching the partitions on the client-side will execute *streak queries* with minimal communication to the server. A streak [BMT17] is defined as a sequence of queries that appear as subsequent modifications of a seed query.

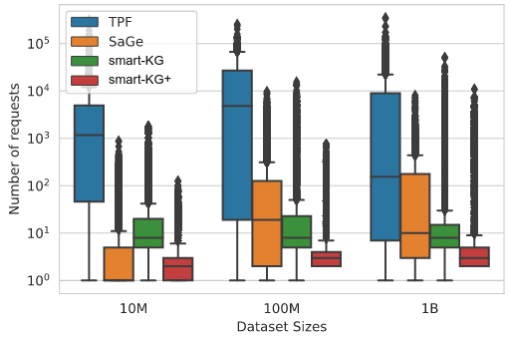
Server CPU Usage. Fig. 6.18c shows that smart-KG, TPF, and smart-KG$^+$ only consume less than 30% of server CPU in order to process the watdiv-sts query workload on all number of clients setups. This is because the aforementioned interfaces limit the client to send certain query patterns (i.e less expressive queries) to the server (e.g. single triple patterns and star patterns). This allows for distributing the query execution computation cost between the client and the server. In contrast, SaGe offers a more expressive server interface with few operators executed on the clients. Thus, SaGe server is able to execute more complex queries which extensively use the server CPU leading to a rapid surge of CPU usage. In particular, SaGe uses less than 30% CPU usage for 1 up to 16 clients and then escalates up to $80 - 100\%$ for 32 to 128 clients.

Server Disk Usage. Table 6.8 presents a comparison of the required disk storage for all compared systems. We consider four KGs with diverse raw data sizes (in N-Triples). In practice, TPF and SaGe rely on the compressed HDT file format that offers a highly space-efficient representation. In turn, smart-KG and smart-KG$^+$ rely on the family partitioning mechanism that demands additional disk space to store HDT partitions, specifically both systems mandate double the N-Triples format size of Watdiv KG. Note that DBpedia requires less storage space since we apply the pruning parameters to reduce the number of materialized HDT partitions. Considering that disk storage is the most economical server resource, smart-KG$^+$ supports an admissible trade-off to obtain better query performance alongside less server CPU consumption.

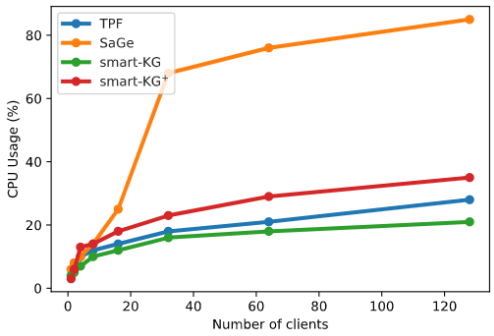
Client CPU and RAM usage. The SaGe client locally performs two main tasks: first, resuming the suspended query execution based on the saved plan received earlier from the SaGe server; second, executing the non-preemptable SPARQL operators including aggregation functions as well as OPTIONAL, ORDER BY, GROUP BY, DISTINCT, etc. Given the aforementioned tasks, SaGe clients, nevertheless, demand a feasible (on



(a) Box plot summary for `transferred data` per query for the 128 clients over `watdiv10M`, `watdiv100M`, `watdiv1B` on `watdiv-sts` workload (log scale)



(b) Box plot summary for the `number of requests` per query for the 128 clients over `watdiv10M`, `watdiv100M`, `watdiv1B` on `watdiv-sts` workload (log scale)



(c) Avg. Server CPU Usage (in %) for increasing number of clients over `watdiv1B` dataset on `watdiv-sts` workload

Figure 6.18: Server resource consumption with increasing number of clients and increasing dataset sizes on `watdiv-sts` workload

Table 6.8: Comparison of storage requirements (in MB) for systems with HDT backend vs original graph size (raw)

Dataset	Raw	family partitioning	*typed*-family partitioning	TPF/SaGe
WatDiv-10M	1,471	2,783	5,632	112
WatDiv-100M	14,876	29,711	58,265	1,186
WatDiv-1000M	151,862	310,574	624,253	12,793
DBpedia	158,197	122,440	150,528	17,904

average 15%) CPU usage and reasonable RAM size ($\sim$ 2GB) for all workloads. In turn, TPF requires a higher computation cost (on average 45%) on the client-side than SaGe since TPF clients locally execute the expensive join operator. Similar to TPF, `smart-KG` performs the join processing of single triple patterns and star patterns queries over the shipped partitions, leading to a higher client CPU consumption (on average 70%) than TPF and SaGe which could be expensive for light client systems. In turn, `smart-KG`$^+$ demands (on average 55%) less client-side processing than `smart-KG`, as it processes fewer intermediate results on the client due to the bind join strategy supported by brTPF as well as the more efficient query plans devised by the server-side optimizer and planner.

Note that the aforementioned percentages de-escalate with an increasing number of clients due to the bottleneck on the server-side since the clients are almost idle awaiting to receive the server response. In other words, the network traffic dominates the query execution of TPF, `smart-KG`, and `smart-KG`$^+$ while context switching overhead and waiting queues dominate in the case of SaGe. Upon comparing the two versions of smart-KG, as depicted in Figure 6.18a, we observe that the `smart-KG` exhibits a higher data transfer and has the potential to consume a greater amount of client RAM compared to `smart-KG`$^+$. Compared to SaGE and TPF, `smart-KG`$^+$ needs a higher client RAM since it loads the HDT partitions in client memory, however it still affordable. For instance, `smart-KG`$^+$ requires up to 3 GB to execute the `watdiv-sts` workload over `watdiv1B`.

6.5.7 Typed-family Partitioning Evaluation

In this part of the evaluation, we focus on evaluating typed-family partitioning using synthetic and real-world KGs on multiple KG sizes and on different query workloads. Therefore, we compare the `smart-KG`$^+$ implementation using only family partitioning and the extended version that additionally uses typed-family partitioning.

6.5.7.1 Typed-family evaluation on the WatDiv dataset

Table 6.9 present a comparison between typed-family partitioning and family partitioning on total transferred data on different sizes of the WatDiv dataset. Typed-family partitioning significantly decreases the number of transferred bytes (on average 27% and 32% over 10M and 100M datasets, respectively) shipped over the network compared to the original family partitioning on `watdiv-sts` workload. As expected, typed-family partitioning demands up to 41% and 46% less transferred data over `watdiv10M` and `watdiv100M`, respectively on `watdiv-stf`$_{bounded}$ and `watdiv-stf`$_{both}$ workloads since we only ship

Table 6.9: Workload Transferred Data per client over watdiv10M and watdiv100M on watdiv-stf$_{bounded}$, watdiv-stf$_{unbounded}$ and watdiv-stf$_{both}$ workloads

Query Workload	Workload Transferred Data (MB)					
	10M			100M		
	Original	Typed	%	Original	Typed	%
watdiv-stf$_{bounded}$	42.14	24.8	(-) 41%	401.97	216,71	(-) 46%
watdiv-stf$_{unbounded}$	28.82	28.85	(+) 0.12 %	236.21	236.24	(+) 0.01%
watdiv-stf$_{both}$	6.02	2.29	(-) 62%	68.61	24.59	(-) 64%
Summary	76.99	55.95	(-) 27%	706.79	477.54	(-) 32%

Table 6.10: Workload Completion Time per client over watdiv10M and watdiv100M on watdiv-stf$_{bounded}$, watdiv-stf$_{unbounded}$ and watdiv-stf$_{both}$ workloads

Query Workload	Workload Completion Time (ms)					
	10M			100M		
	Original	Typed	%	Original	Typed	%
watdiv-stf$_{bounded}$	22956	12621	(-) 45%	47167	27479	(-) 42%
watdiv-stf$_{unbounded}$	10538	11237	(+) 7%	12228	12691	(+) 3%
watdiv-stf$_{both}$	7668	3702	(-) 52%	15022	5941	(-) 60%
Summary	41162	27560	(-) 33%	74417	46111	(-) 38%

the family partitions that contains the exact solution bindings to the star-shaped subquery. We also show in Table 6.10 the impact of typed-family partitioning on the watdiv-sts workload completion time. Typed-family partitioning has substantially reduced (over 40%) the completion time for watdiv-stf$_{bounded}$ and watdiv-stf$_{both}$ on both watdiv10M and watdiv100M datasets.

In Fig. 6.19 and Fig. 6.20, we show at the query level the impact of typed-family partitioning on the execution of different query shapes extracted from the WatDiv Basic Testing query set. Fig. 6.19 shows that typed-family partitioning significantly decreases the data transferred of watdiv-btf$_{bounded}$ workload queries. For instance, using typed-family partitioning, query S3 requires only 3% and 1% of the transferred data required over watdiv10M and watdiv100M, respectively, in comparison to using original family partitioning. In addition, when using typed-family partitioning, queries F1 and S2 demand 97% - 99% less data transfer than when using family partitioning. Note that typed-family partitioning has no influence on the queries of the watdiv-btf$_{unbounded}$.

Fig. 6.20 shows the execution time of the workloads. For the watdiv-btf$_{bounded}$ workload, the execution time of queries has been significantly reduced thanks to typed-partitioning for downsizing the shipped partitions, e.g., with a percent of decrease between -30% and -60 % in watdiv-100M dataset. Yet, typed-family partitioning has a positive bearing on the query performance of the watdiv-btf$_{bounded}$ workload for several queries. For the watdiv-btf$_{unbounded}$ workload, the runtime with typed-family partitioning has a slight increase of 5-8 ms for unbounded queries. This increase is attributed to the query planner needing to search through additional metadata associated with typed family partitioning. Yet, this delay is an implementation detail that can be optimized further.

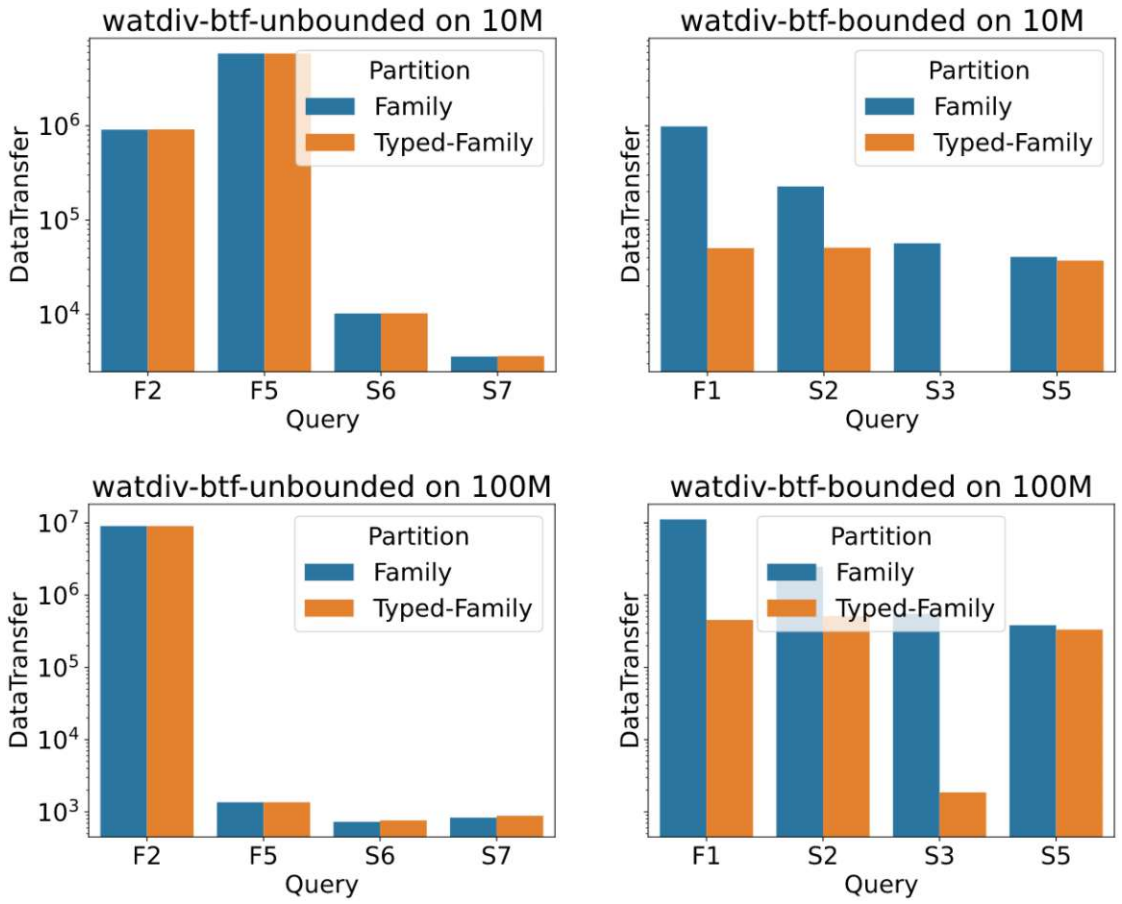


Figure 6.19: Data transferred per query (in bytes) over `watdiv10M` and `watdiv100M` on `watdiv-btf`$_{unbounded}$ and `watdiv-btf`$_{bounded}$ workloads

6.5.7.2 Typed-family evaluation on the DBpedia dataset

In this section, we analyze the impact of typed-family partitioning compared to family partitioning on the execution time of the `DBpedia-btt`$_{bounded}$ and workload. Note that `smart-KG` relies on family partitioning where we do not materialize any partition that contains the predicate `rdf:type` since `rdf:type` $\notin P'_G$.

In Fig. 6.21, we divide the queries in `Dbpedia-btt`$_{bounded}$ into four different categories based on the number of star-shaped subqueries, subquery selectivity, and a star-shaped query combined with single triple patterns. Fig. 6.21a shows the performance for highly selective star queries. `smart-KG`$^+$ with typed-family partitioning executes queries Q1 up Q4 slightly slower than `smart-KG`$^+$ with family partitioning. This is due to the fact that family partitioning does not materialize any partitions that contain `rdf:type` predicate, since the percentage of triples with this predicate is higher than the defined threshold

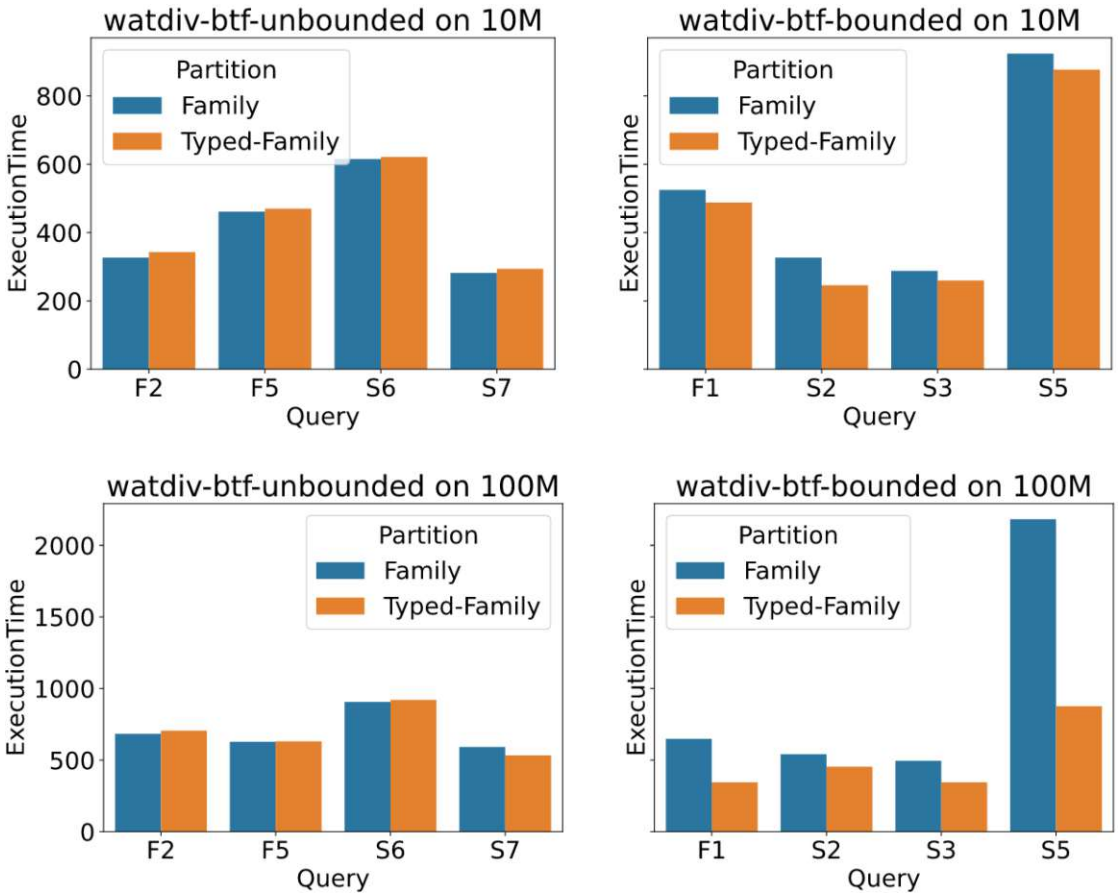
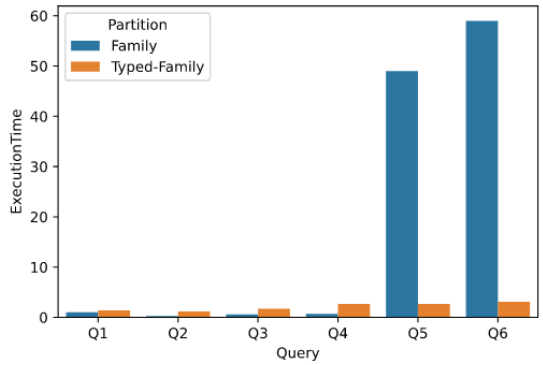


Figure 6.20: Execution time per query (in ms) over `watdiv10M` and `watdiv100M` on `watdiv-btf`$_{unbounded}$ and `watdiv-btf`$_{bounded}$ workloads

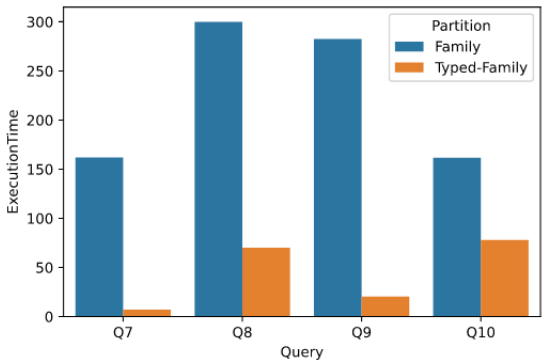
$\tau_{p_{high}}$. In queries Q5 and Q6, `smart-KG`$^+$ with typed-family partitioning achieves a better performance since it ships a partition that resolves the query locally while brTPF has a poor performance since some of the triple patterns are non-selective (even though the entire star-query is highly selective).

Fig. 6.21b shows that relying on typed-family partitioning achieves better query processing performance compared to family partitioning. This is because `smart-KG`$^+$ ships a typed partition that contains the solution mappings of the entire star query, while on using family-based partitioning, `smart-KG`$^+$ will utilize brTPF to resolve the triple pattern with *rdf:type*, which require an enormous number of requests to join with a non-selective star subquery.

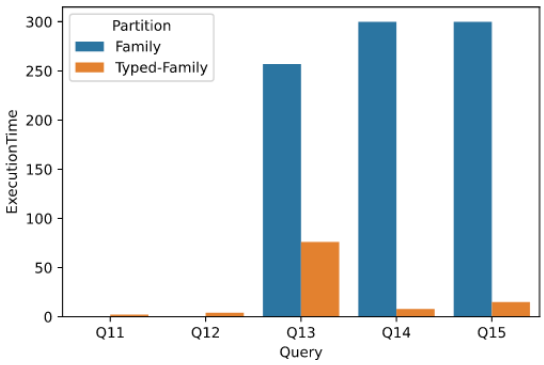
Fig. 6.21c presents the execution time of queries that combine a star subquery with a couple of single triple patterns in a BGP. In queries Q11 and Q12, `smart-KG`$^+$ with



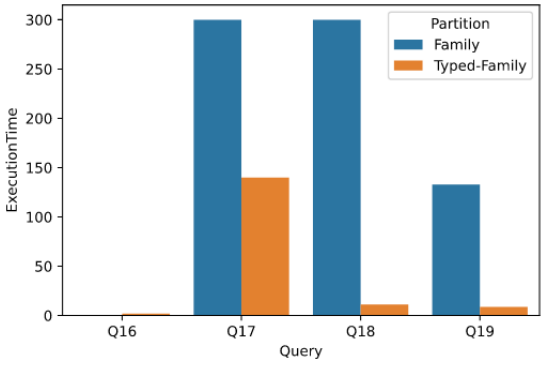
(a) Execution time for queries (in sec.) with a high selective single star subquery in DBpedia-btt$_{bounded}$ workload



(b) Execution time for queries (in sec.) with a low selective single star subquery in DBpedia-btt$_{bounded}$ workload



(c) Execution time for queries (in sec.) with a single star and multiple triple patterns in DBpedia-btt$_{bounded}$ workload



(d) Execution time for queries (in sec.) with multiple stars in DBpedia-btt$_{bounded}$ workload

Figure 6.21: Execution time per query (in sec) on DBpedia-btt$_{bounded}$ workload

family partitioning performs slightly better than with typed-family partitioning, since these queries include highly selective triple patterns and do not require shipping an entire partition to resolve the query. On the other hand, Q13, Q14, and Q15 show a significant improvement when using typed-family partition since it reduces the amount of data transferred.

Fig. 6.21d shows that smart-KG$^+$ has a better performance in queries Q17, Q18, and Q19 when relying on typed-family partitioning and better performance in query Q16 when using family partition. Note that the query performance highly depends on the selectivity of the star typed subquery Q_s (i.e the size of the shipped partition in case of typed-family partitions) compared to the selectivity of Q'_s after decomposing the typed star to Q'_s and Q''_s. Finally, in queries with no bound types, the observed performance of family partitions and typed-family partitions is almost identical. This is consistent with the results on WatDiv watdiv-btf$_{unbounded}$ presented in Section 6.5.7.1.

Table 6.11: Impact of Typed-Family Partitioning on WiseKG's Performance on watdiv10M dataset (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in milliseconds, TO: Timeouts. ET is in milliseconds, DT is in MB

Workload	WiseKG^{Family}				WiseKG$^{Typed-Family}$			
	Req	DT	ET	TO	Req	DT	ET	TO
watdiv-sts	2452	101.81	39610	6	**2301**	**85.72**	**39093**	6
watdiv-btf	179	2.73	23666	1	179	27.32	23680	1
watdiv-stf$_{bounded}$	528	22.54	17097	0	**377**	**10.20**	**12285**	0
watdiv-stf$_{unbounded}$	36	9.0	3286	0	36	9.0	3235	0
watdiv-stf$_{both}$	98	5.24	2787	0	98	**1.49**	**1970**	0

6.5.7.3 Assessing the impact of typed-family partitioning on WiseKG

WiseKG [AAM+21] is an LDF interface that dynamically shifts the query processing load between client and server. WiseKG combines two LDF APIs (SPF and smart-KG) that enable server-side and client-side processing of star-shaped sub-patterns. WiseKG decides whether the star-subqueries should be processed on the client or on the server. For this, WiseKG relies on a cost model that picks the best-suited API per sub-query based on the current server load, client capabilities, estimation of necessary data transfer between client and server, and network bandwidth. By leveraging this cost model, WiseKG dynamically distributes query processing tasks between servers and clients, better-utilizing server resources and maintaining high-performance levels even under conditions of heavy load.

Earlier experiments have demonstrated that WiseKG outperforms state-of-the-art stand-alone LDF interfaces, especially under highly demanding workloads. Consequently, this section evaluates the impact of our typed-family partitioning on WiseKG's performance. To do so, the following two versions of WiseKG are developed:

- WiseKG^{Family}: The original version of WiseKG relies on the family generator from smart-KG.
- WiseKG$^{Typed-Family}$: An extension of the earlier version where we incorporate typed-family partitioning proposed in smart-KG+.

The experimental results, as presented in Table 6.11, are based on watdiv10M dataset. We observe that WiseKG$^{Typed-Family}$ achieves significant reductions in data transfer by 16%, 54%, and 71% for watdiv-sts, watdiv-stf$_{bounded}$, and watdiv-stf$_{both}$, respectively, when compared to WiseKG^{Family}. Additionally, WiseKG$^{Typed-Family}$ requires 7% fewer requests than WiseKG^{Family} for watdiv-sts and 28% fewer requests for watdiv-stf$_{bounded}$. This performance improvement is attributed to the adoption of typed-family partitioning, which effectively reduces data transfer and the number of requests for queries involving bounded star-typed patterns. It is worth noting that WiseKG's performance remains unaffected by typed-family partitioning in the case of the watdiv-btf$_{unbounded}$ workload, consistent with the results presented in Section 6.5.7.1. Moreover, the impact of typed-partitioning on WiseKG's performance with

the watdiv-btf workload is minimal due to its relatively smaller size query workload, and the cost model of WiseKG effectively executes most of the queries on the server-side using the SPF API.

6.6 Lesson Learned

Concluding the evaluation of the experimental results for our proposed approach, we provide a summary of our lessons learned in the following:

Ablation Study

- **Server-side query planning (NP) enhances complex query performance**: The proposed solution (brTPF+NP) enhances the performance of complex queries (C) by up to 50% through effective star reordering. However, our solution encounters timeouts in some cases, mainly because of attaching large intermediate results back to the server and query planning mismatches.
- **Trade-off in query planning strategies**: The new server-side query planner in smart-KG+ achieves better query plans, but this improvement comes at the cost of increased server requests. Additionally, the strategy to query brTPF achieves better performance in queries requiring a small number of intermediate results, while querying TPF is more efficient for queries involving many intermediate results.

System Performance Evaluation

- **Improved throughput, resource consumption and scalability**: smart-KG+ scales better, achieving higher query throughput with an increasing number of clients over various graph sizes. Our solution requires fewer transferred data and sends a lower number of requests per query compared to other systems. Additionally, it performs efficiently on different graph sizes, outperforming TPF and SaGe in workload completion time, achieving up to 18x and 7x faster performance with increasing the number of concurrent clients. In summary, smart-KG+ demonstrates efficient resource consumption and scalability.
- **Query shape matters**: smart-KG+ excels in complex queries (C), but it may not be as efficient in simple queries (L) and moderately selective queries (F).

Typed-family Partitioning

- **Typed-Family partitioning reduces data transfer**: The evaluation shows that typed-family partitioning significantly decreases the amount of data transferred over the network compared to using only family partitioning. On average, using typed-family partitioning results in a reduction of 27% and 32% in transferred

bytes for 10M- and 100M-sized datasets, respectively, in the watdiv-sts workload. This reduction can be even higher (up to 46%) in certain cases, such as watdiv-stf$_{bounded}$ workload.

- **Query completion time improvement**: Typed-family partitioning substantially reduces the completion time for queries in the watdiv-stf$_{bounded}$ workload on both 10M- and 100M-sized datasets. The improvement in completion time ranges from 40% to 46% for these specific workloads.
- **Minimal impact/overhead on unbounded queries**: For queries without bound types (unbounded queries), the observed performance of family partitions and typed-family partitions is almost identical.
- **Extension to WiseKG**: The evaluation also extends the evaluation to WiseKG, an LDF interface that dynamically shifts query processing load between clients and servers. The results show that typed-family partitioning significantly reduces data transfer and the number of requests of WiseKG.

6.7 Summary and Limitations

We introduced smart-KG$^+$, a hybrid shipping approach to efficiently query Knowledge Graphs (KGs) on the Web, while balancing the load between servers and clients. We combine the Bindings-Restricted Triple Pattern Fragment (brTPF) strategy with shipping compressed graph partitions that can be locally queried at the client. The served partitions are based on predicate and typed-families benefiting from the special nature of the rdf:type predicate. In smart-KG$^+$, we implement a server-side query planner to provide accurate plans tailored to consider the two execution shipping partition and brTPF based on cardinality estimations.

Our evaluation shows that smart-KG$^+$ performs on average 10 times faster, use 5 times less network traffic, and sends 20 times fewer requests, with 5 times less server CPU, outperforming the state-of-the-art approaches. We show an extensive experimental study on synthetic and real datasets that, at the cost of reasonable server disk storage, smart-KG$^+$ improves the execution time of the query workloads and reduces network cost.

Our proposed interface,smart-KG$^+$, requires a request to the server-side query planner with an average processing time of 70 milliseconds. However, client-side query planner requires an initial data transfer of 1.75MB, including crucial metadata for identifying the queried knowledge graph partition. Though the metadata file's shipping time takes around 700 milliseconds, caching it locally improves subsequent query processing. Notably, our results assume the metadata is already available on the client-side.

In low-selective queries with large intermediate results, brTPF performs worse than TPF due to higher intermediate results attachment costs. Our proposed server-side query

planner mismatch may cause longer execution times, but the query planner generally provides more efficient execution plans. On the other hand, brTPF outperforms TPF for queries with a small number of intermediate results, while TPF is better for queries with more intermediate results. Future work includes implementing a cost model to decide between TPF, brTPF, or shipping a vertical partition for single triple pattern execution. Finally, our new version of `smart-KG`$^+$ provides better performance than `smart-KG`, delivering a better total geometric mean for requests, data transfer, and execution time.

A recent study [RHSG14] shows that a tiny portion of the entire KG is actually accessed by a typical DBpedia query workload. Thus, as future work, we plan to consider the query workload during the KG partitioning to minimize the number of materialized partitions and focus on the ones required for the query load [MAA18, HS13, GHS14]. In addition, we plan to investigate an online family partitioning mechanism based on the current query workload [AAK$^+$16].

Our future work includes exploring other partitioning strategies to reduce the network traffic since this is one of the main factors impacting the performance as shown in our experiments. For this, we plan to generalize the cost model introduced in WiseKG [AAM$^+$21] to dynamically delegate the query processing load between servers and clients more effectively. Note that, the typed-family partitions and the query planner presented in this work can already be incorporated into WiseKG to enhance its performance when evaluating typed star queries. We plan to introduce update strategies for the constructed partitions to manage evolving KGs. Lastly, we plan to investigate the integration of partition-based LDF interfaces in the landscape of heterogeneous KG federations [HA22].

CHAPTER 7

Partition-based Linked Data Fragments: Alternatives

This chapter provides an analysis of partitioning techniques that have been employed in both centralized and distributed RDF processing and their relevance in enabling efficient Web querying and as a possible alternative for partition-based Linked Data Fragments. To achieve this objective, we formalize existing RDF partitioning techniques, such as horizontal and vertical partitioning, as possible shipping strategies for partition-based LDF interfaces. Our analysis is based on surveys of relevant literature including [KM15, AHKK17, ASY+21]. Additionally, we present a summary of graph partitioning techniques and RDF systems used in the literature, which is presented in Table 7.1.

7.1 Vertical Partitioning (VP)

VP [AMMH07] creates a partition for each unique predicate in $pred(G)$. i.e., in our terms,

$$\mathcal{G} = \{G_p \mid p \in pred(G) \wedge G_p = \{\omega((?s, p, ?o)) \mid \omega \in [\![(?s, p, ?o)]\!]_G\}\}$$

Next, *admissible queries* are any single triple pattern queries $Q = \{tp\}$ where

$$\sigma(G, Q) = \{G_p \in \mathcal{G} \mid p = pred(Q) \cap pred(G) \vee pred(Q) \text{ is a variable}\} \tag{7.1}$$

That is, for any triple pattern query Q, either a single predicate partition corresponding to the query predicate, or all predicate partitions would be returned.

Many RDF processing systems (cf. Table 7.1) report achieving a high query performance using vertical partitioning. Yet, as a base partitioning mechanism for partition-based

Table 7.1: An overview of the exiting graph partitioning techniques utilized in RDF engines

Partitioning Techniques	**RDF Systems**
Vertical Partitioning	SW-Store [AMMH09], PRoST [CFL18], S2RDF [SPSL16], PigSPARQL [SPL11], SPARQLGX [GJGL16], SANSA [LSB+17], Sempala [SPNL14], SparkRDF [CCZZ14], Jena-HBase [McB01], CliqueSquare [DGK+15], HadoopRDF [DWNY12]
Horizontal Partitioning	AllegroGraph[1], Blazegraph[2], SHARD [RS10], DiS-tRDF [WMPH19], Partout [GHS14], Akhter et. al [ANS18]
Hash Partitioning	YARS2 [HUHD07], TriAD [GSMT14], AdPart [AAK+16], PigSPARQL [SPL11], CliqueSquare [DGK+15], Koral [JST17], CumulusRDF [Har11a], SHAPE [LL13], SHARD [RS10]
Workload-aware Partitioning	Partout [GHS14], chameleon-db [AÖDH13], WARP [HS13], WORQ [MAA18].
K-way Partitioning	Akhter et. al [ANS18], EAGRE [ZCTW13], H-RDF-3X [HAR11b], TriAD-SG [GSMT14]

LDFs this approach only works well for triple pattern queries with bounded predicates, whereas for any triple patterns with unbound predicates, all partitions would need to be shipped. Along these lines, assuming all predicates in Q are bound, a strict lower bound for the number of shipped partitions is $|pred(Q) \cap pred(G)|$, because you only need to ship partitions for predicates mentioned in the query, that also occur in G. As a second drawback of using vertical partitioning in the context of partition-based LDF is that it only supported single triple queries, any joins or more complex patterns would need to be fully evaluated on the client side. Also, full vertical partition shipping has potential downsides compared with TPF or brTPF, which solves any binding in triple patterns directly on the server side. For all these reasons, we will in our proposed approach rather use (br)TPF directly for single triple queries.

7.2 Horizontal/Range/Sharding Partitioning

In the context of distributed relational databases, horizontal partitioning involves splitting a relation horizontally, i.e. row-wise, into sub-relations based on selections to enhance the load balancing. Analogously, RDF management systems have adopted horizontal partitioning strategies to distribute the triples of an RDF graph into multiple partitions based on certain selection criteria. In these strategies, the selection is typically used to generate horizontal subsets of the RDF triples for very common predicates (such as e.g. `rdf:type`, which often does not lend itself well to vertical partitioning techniques), where each subset consists of all the triples that satisfy a predetermined selection condition on

the objects or subjects. Herein, we exemplify horizontal partitioning based on object ranges; that is, if we assume partitions per n object ranges (e.g. from a histogram) into a set of ordered values $\{v_0, \ldots, v_n\}$. Given the RDF model, this is not an unreasonable assumption, indeed, both literals and likewise URIs could be assumed to be ordered with respect to their string representations, and – even if many real-world RDF graphs do not contain blank nodes – also blank nodes could, while not ordered in the RDF model itself, be canonicalised [Hog17] and ordered, respectively. Accordingly, we can define

$$\mathcal{G} = \{G_i \mid 1 \leq i \leq n \wedge G_i = \{\omega((?s, ?p, ?o)) \mid \omega \in [\![(?s, ?p, ?o)\ \texttt{FILTER}\ (v_{i-1} < ?o \wedge ?o \leq v_i)]\!]_G\}\} \tag{7.2}$$

Object-based horizontal partitioning could be used for partition shipping, where any BGP query Q is admissible that consists of triples with the same object, i.e., $obj(Q) = \{o\}$ (which of course includes single triple queries with bounded object), but, again, for unbounded objects, the entire partitioning $\mathcal{G}$ would need to be shipped:

$$\sigma(G, Q) = \{G_i \in \mathcal{G} \mid (v_{i-1} < o \wedge o \leq v_i) \vee o \text{ is a variable}\} \tag{7.3}$$

Horizontal partitioning could be analogously defined for bound subjects, or be combined with vertical partitioning (i.e. be used to further subdivide vertical partitions); in fact, vertical partitioning as defined above could be viewed as a "special form" of horizontal partitioning on the predicate position, with "predicate ranges" corresponding to the single predicates in $pred(G)$.

Variations of horizontal partitioning have been used successfully by several RDF systems, especially in distributed environments (cf. Table 7.1), where partitions are allocated to different nodes while minimizing the communication cost among the nodes (by placing jointly queried data together) and balancing the node workload (by placing highly requested partitions in different nodes). In general, horizontal partitioning supports efficient querying for queries that require shipping a single partition based on the FILTER condition that defines the shipped partitions. As such, there are similar (dis-)advantages as for vertical partitioning: for our example of horizontal partitioning on the object, whenever the object is unbound, all partitions would need to be retrieved. Likewise, depending on the choice of ranges (v_1 to v_n) to "split" the partitions and data distribution, the matching partitions could contain potentially large amounts of irrelevant data or different horizontal partitions could contain a prohibitively large superset of the answers of the query, e.g., by including further predicates which are not requested in the query. The latter could be remedied by combining more sophisticated forms of vertical and horizontal partitioning. For example, family-based partitioning techniques described in Section 4.1 and Section 6.1 can be seen in a sense as vertical partitioning and a combination of vertical and horizontal partitioning, respectively..

7.3 Hash Partitioning (HP)

Hash-based partitioning is a common partitioning strategy among RDF distributed systems. For instance, *position-based hashing* is a lightweight partitioning strategy that applies a hash function to a particular position (e.g. subject-based hashing) in triples, distributing the RDF triples according to their hash values into a fixed number of n bins. Thus, all the triples with the same value in this position (e.g. same subject) are allocated to one partition. Hash partitioning is computationally inexpensive plus the hash operation can be efficiently computed in parallel. However, as usual with hashing, hash collisions may cause skewed partition sizes. Hash-based partitioning could be defined in a very similar manner as above, exemplified here for subject-based hashing with n partitions. Assuming a suitable hash function $h(\cdot)$ and the modulo operator being available in SPARQL:

$$\mathcal{G} = \{G_i \mid 1 \leq i \leq n \wedge G_i = \{\omega((?s, ?p, ?o)) \mid \omega \in [\![(?s, ?p, ?o) \texttt{ FILTER } (h(?s) = i)]\!]_G\}\} \tag{7.4}$$

For position-based hashing (analogously to position-based horizontal partitioning explained above), any basic graph patterns sharing the same value in the respective position, e.g. subjects, would be admissible patterns. For such admissible queries $\sigma(G, Q)$ could again be analogously defined based on the hash function $h(\cdot)$ of the resp. position, that is e.g. based on $h(subj(Q))$, as above, with the same problems of retrieving all $\mathcal{G}$ whenever the subject is unbound. Likewise, these definitions can easily be extended to object, predicate, or even triple-based hashing (based on a "ternary" hash function $h(s, p, o)$).

Position-based hashing can be extended by specific hash functions, e.g. prefix-hashing [JST17], to ensure that subjects (or other position terms) with the same prefix end up in the same partition, which can be exploited in range queries. Another extension is k-hop hashing which could cater for certain path queries, by creating (potentially overlapping) partitions that extend simple hash-based partitions with the k-hop neighborhoods of the hashed triples [LL13].

7.4 Workload-aware partitioning

Workload-aware partitioning makes use of query workloads in order to partition RDF graphs. Ideally, the query workload includes representative queries extracted from a real-world or a synthetic/simulated query log.

Several RDF distributed systems rely on workload-aware partitioning such as Partout [GHS14], chameleon-db [AÖDH13], WARP [HS13], and WORQ [MAA18]. Bonifati et. al [BMT17] has conducted an analytical study of end users' queries harvested from real-world query logs of SPARQL endpoints. According to the analysis of the graph structure of queries, tree-like shapes such as single triple patterns, chains, stars, trees, and forests are the most observed shapes. We consider the aforementioned observation especially star queries in family partitioning technique introduced in Sec. 4.1.

In our context, workload-aware partitioning could be seen as a form of “caching”, where subgraphs containing a superset of or exactly the results of particularly common subqueries could be stored as separate partitions. However, in order to make use of such caching, complex queries would need to be analyzed whether they contain any of these "cached" subqueries or respectively subqueries subsumed by the cached queries. Since such a form of partitioning is rather related to index-learning from query logs, a concrete formalization depends on formalizing/extractable common query patterns from such query logs. we see various options here and consider them as somewhat complementary and orthogonal to our current work. In the present paper, we restrict the scope to partitioning definable by the (characteristics of the) graph only. We therefore leave a concrete formalization/implementation of partition-based LDF following this idea to future work.

7.5 K-way Partitioning (KP)

Similarly, K-way partitioning is not directly amenable to our framework: K-way partitioning algorithms, such as [KK98] strive to partition the graph into roughly equal-sized smaller graphs with the intention of minimizing the number of edges linking vertices from different partitions and thus could be viewed rather as a “clustering” technique for RDF graphs than partitioning based on/or specifically used for evaluating particular query patterns. As such, we also leave it open for future work on how/whether such techniques could be used for computing a partitioning $\mathcal{G}$ that allows deriving an easy-to-compute selector function σ.

CHAPTER 8

Reproducibility

Reproducibility is a fundamental principle of scientific research [CK92, har12], as it allows different research groups to verify and build upon the findings of previous studies. Reproducible research means that the same results can be obtained by different researchers using the same methods and data [GFI16]. However, reproducibility has become a challenge in many fields of science, including life sciences [BE12], psychology [AAA+15], and computational science [RRR22, CHI15].

In this chapter, we utilize Docker, one of the newest and fastest emerging DevOps tools, to achieve better reproducibility for our research work [Boe15]. Docker[1] is a tool that enables software engineers, system admins, and researchers to deploy their software in isolated environments (i.e. containers) that run directly on the host OS (i.e. Linux) rather than virtual machines that run on a guest OS, requiring a substantial computational cost. The primary utility of Docker [Ber14] is to package a software application with its dependencies into a standardized self-contained component named containers. In essence, Docker enables us to (i) document our client-server environment, and (ii) consistently and uniformly distribute and execute our software across diverse infrastructures. This approach can help to ensure the reproducibility of our research results and facilitate collaboration with other researchers.

To this end, we provide reproducible experiments in the Web querying research field through building a DevOp approach that facilitates: (i) providing all configuration parameters for both our introduced systems and the compared state-of-the-art systems; (ii) reporting the operating systems, client, and server specifications, and network configurations, and (iii) making all RDF KGs and their partitions, query workloads, system monitoring tools, and source codes available to allow researchers to reproduce our experiments.

[1] Docker: `https://www.docker.com/`

8.1 Linked Data Fragments Implementation

The first instantiation of Linked Data fragments is the Triple Pattern Fragment (TPF), where both client and server implementations for web SPARQL queries have been open-sourced[2]. The TPF implementation has been employed to evaluate and implement various extensions to the TPF interface [HVMdW15, FSMM15, TVCM16, SVH+15, HVV+15, TSVM17, HA16]. For the TPF client, the Java framework Jena has been adapted, while the server is implemented in Java servlets with HDT as the back-end. Like many other successive Web interfaces such as brTPF and SPF, we have used this source code as the core for our implementation for both `smart-KG` and `WiseKG`. The following describes the major extensions that have been made to both the client and server implementations.

`WiseKG` Server Implementation: In the following, we will focus on the server implementation of `WiseKG`, since it includes by design brTPF, SPF, and `smart-KG`. As the basis for our `WiseKG` server, we used an existing Java servlet implementation of the brTPF interface and extended it with two inherited servlets one for SPF interface, and another `smart-KG` interface, plus a servlet to support the server query planner. The result is that all interface implementations are inherited from the same Java servlet, which selects which of them to invoke according to the HTTP GET request it receives: (i) if the request requires a full query plan, the query planner servlet is used to generate a query plan based on the cost model as the response; (ii) if the request contains the brTPF selector, the brTPF implementation provides the response; (iii) if the request contains the SPF selector, the SPF implementation is used to generate the response; or (iv) if the HTTP request contains only a `smart-KG` selector, the `smart-KG` implementation is used. Having all implementations in a single consolidated software component has the advantage that: (i) frequently executed basic functions such as the query parser, decomposer, executor, and RDF result serializer are reused, and (ii) experimental results are therefore not affected by potential implementation variations in such basic functions.

The `WiseKG` server returns an HTTP response to the client containing either (i) a Gson object describing the query plan of the requested query; (ii) RDF triples from evaluating a triple pattern partitioned into pages based on the brTPF interface; (iii) RDF triples from evaluating a star pattern partitioned into pages based on the SPF interface; or (iv) HDT partitions from evaluating a star pattern based on the smart-KG interface.

`WiseKG` Client Implementation: We implement `WiseKG` client based on the Java implementation of brTPF client. We adapt the query evaluation algorithm of brTPF [HA16] in the `WiseKG` client implementation (see Alg. 5.2) to evaluate SPARQL BGPs. This algorithm is based on a dynamic pipeline of iterators. The query results are evaluated recursively by executing the pipelines where each pipeline is generated for a sub-query resulting from the query decomposition of the initial BGP. Each iterator generates the solution mappings of a sub-query and chunks it to the next iterator. The algorithm uses the cost model estimates to dynamically determine the join order of the sub-queries, where sub-queries with a lower cost estimate are executed first.

[2] `https://github.com/LinkedDataFragments/`

To this end, by extending the original TPF framework, we ensure the comparability in our experimental results and enable a fair comparison between the state-of-the-art approaches including TPF, brTPF, and SPF. In this way, we have unified the storage back-end (HDT), programming language, and basic query evaluation components with the state-of-the-art approaches with one exception: SaGe, since it follows a completely different stack.

In the following, we detail the commands required to install and execute `WiseKG` client and server[3]:

Command 1. *Both `WiseKG` client and server are written in Java 8. We can create a JAR/WAR (we follow the same steps as the LDF framework for deployment) which is as follows:*

```
mvn␣install
```

The server can run with Jetty from a single jar as follows:

```
java␣-jar␣wisekg-server.jar␣[config.json]
```

[config.json] The config file describes the data sources that will be available on this server. An example of the config file is available online[a]

To use the client, we use the following command:

```
java␣-jar␣[filename].jar␣␣false␣-f␣[Server]␣-q␣[Query␣File]
```

[Server]: This argument takes the server URL. [Query File] This argument receives the input query to be executed.

Here is an example of executing a query on `WiseKG`:

```
java␣-jar␣wisekg.jar␣-f␣kg-server.ai.wu.ac.at:8084/sparql/dbpedia
-q␣sparqlExample.sparql
```

sparqlExample.sparql: the address to a file containing a single SPARQL query.

[a] `https://github.com/LinkedDataFragments/Server.Java/blob/master/config-example.json`

8.1.1 Partition Generation

The first step to using `smart-KG` or `WiseKG` is to generate the KG partitions. This KG partition generation tool can be used either by compiling and installing the source code

[3] WiseKG `https://github.com/WiseKG`

or by using the provided Docker version[4]. All partitions generated in our experiments have been made publicly available and can be accessed via this link[5].

In the following, we detail the commands required to execute family-based partitioning introduced in Equation 4.5 and Algorithm and 4.1 and typed family-partitioning introduced in Equations 6.1 and 6.7 in order to generate the KG partitions required for both `smart-KG` and `WiseKG` Web interfaces.

Command 2. *We show basic statistics on the input dataset such as the total number of triples, unique predicates, initial families, and merged families using the following command:*

On using the native version:

```
./hdt-cpp-molecules/libhdt/tools/getFamiliesEstimate␣[arguments]␣<hdtfile>
```

- Upon completion of the compilation and installation process, the partitioning tools are situated in the directory "/hdt-cpp-molecules/libhdt/tools".

Or on using the docker version:

```
docker␣run␣-v␣/host/path/target:/file/path/within/container␣--rm
smartkg-creator␣getFamiliesEstimate␣[arguments]␣<hdtfile>
```

-v: This parameter enables the mounting of a volume driver onto the Docker image. It requires a path to a directory containing the HDT file to be partitioned. The path provided refers to the location of the files on the local machine, which is represented as "/host/path/target". This is followed by a colon ":" and the path within the container where the HDT files will be mounted, such as "/file/path/within/container".

As shown, the command accepts the input RDF dataset in HDT format. In the following, we list possible arguments for getFamiliesEstimate:

- *-S: This parameter enables the selection of only those families that have a minimum percentage of subjects present in the dataset. This option is particularly useful for unstructured datasets like Dbpedia. By default, the minimum percentage is set to $\alpha_s = 0.01$. To modify this value, users need to invoke the "-P" parameter.*
- *-P: This parameter allows users to set the minimum percentage of subjects required for a family to be selected. This parameter can only be used in conjunction with the "-S" parameter. If the "-S" parameter is enabled and "-P"*

[4]Details of installation are provided in the README. `https://github.com/smart-KG/smartKG-creator-types`

[5]`DataPartitions:https://smartkg-data.cluster.ai.wu.ac.at/`

is not specified, the default value of 0.01 is used. The value specified for this parameter denotes α_s in equation 4.7.

- *-L <percentage>: This parameter allows users to specify the percentage of infrequent predicates in terms of their occurrences within the dataset. Predicates that have less than the specified percentage of occurrences (as a percentage of the total number of triples) will be discarded and not considered in the families. The default value of this parameter is 0.01%. The value specified for this parameter corresponds to τ_l in equation 4.6.*
- *-H <percentage>: This parameter allows users to specify the percentage of occurrences at which massive predicates are to be cut, expressed as a percentage of the total number of triples. Predicates that have more than the specified percentage of occurrences will be discarded. The default value of this parameter is 0.1%. The value specified for this parameter corresponds to τ_h in equation 4.6.*
- *-m <Percentage>: This parameter enables users to specify the maximum size of a new group in terms of a percentage of the total number of triples. For instance, if the parameter is set to 5, a new group is created only if the estimated size is less than 5% of the total number of triples. If "-m" is set to 100, then all groups are allowed. The value of this parameter corresponds to α_t in equation 4.8.*
- *-q: activate quick estimation (do not perform grouping)*

Example 9. *This example is to demonstrate the Command 2 based on RDF Friends graph:*

```
./hdt-cpp-molecules/libhdt/tools/getFamiliesEstimate friends2023.hdt
```

The output is as follows:

Total predicates: 16

Total triples: 79

tau_l min pred:0.01

tau_h max pred:1.0

Number of families:10

Original families: 10

Total potential families groups: 8

Final number of families:18

Time total 2 ms 74 us

Command 3. *We generate the family-based partitions of an input dataset using the following command which is the implementation of Equation 4.5 and Algorithm 4.1 and also we generate the typed family-based partitions of the input dataset using the following command which is the implementation of Equation 6.7 and Equation 6.6:*

On using the native version:

```
./hdt-cpp-molecules/libhdt/tools/getFamilies␣[arguments]␣<hdtfile>
```

Or on using the docker version:

```
docker␣run␣-v␣/host/path/target:/file/path/within/container␣--rm
smartkg-creator␣getFamilies␣[arguments]␣<hdtfile>
```

Typed-family partitioning generation process accepts RDF dataset in HDT format. In the following, we list possible arguments:

- *we use the arguments (-S, -P, -L, -H, -m and -q) for the exact functions as Command 2.*
- *-s <splitFilePrefix>: This argument allows users to partition triples based on existing families that are described in the specified JSON file with the given prefix.*
- *-e <exportFile>: This argument exports the metadata of families in <exportFile>.json and the groups in <exportFile>_group.json. This information can be used by the query planner to locate the HDT partition containing the results of a given query.*
- *-i: This argument includes infrequent predicates with occurrences less than the user-defined threshold τ_l (default 0.01%), which may result in the creation of more partitions. This argument is set to false by default.*
- *-C <classesFile>: this argument accepts a file containing a list of classes separated by a new line. The typed-family partition is applied only to the classes listed in this file. This argument is used only to perform typed-family partitioning as defined in Section 6.3.*

- *-c: This argument cuts massive predicates, i.e., predicates with occurrences greater than or equal to the user-defined threshold τ_h (default 0.1%), resulting in the creation of fewer but larger partitions. This argument is set to false by default and cuts the predicates according to Equation 4.6.*
- *-d: This argument dumps infrequent predicates into a dedicated JSON file with the prefix "_infreqPreds". Infrequent predicates are defined by Equation 4.6.*
- *-u: ungroup – This argument performs family partitioning without the grouping step, which generates partitions based solely on the original families defined in Equations 4.2 and 4.3.*
- *-G: This argument exports each family into a separate JSON file.*
- *-v: This argument enables verbose mode, providing detailed results by printing all triples during partitioning. We recommend using this argument only for testing purposes.*
- *-h: This argument provides a verbose explanation of the available arguments.*

In the following, we detail an example of generating the family-based partitions of the Friends RDF graph from Example 7:

Example 10. *(From RDF KG to family KG partitions)*

The initial step involves converting the RDF file into HDT format, which can be executed through ***libhdt*** *library. This library, an integral part of the partition generation tool, offers a range of functionalities provided by HDT*[a]*. This conversion step can be performed using the following command:*

On using the native version:

```
./hdt-cpp-molecules/libhdt/tools/rdf2hdt␣friends.ttl␣friends.hdt
```

Or on using the docker version:

```
docker␣run␣-v␣/host/path/target:/file/path/within/container␣--rm
rdf2hdt␣friends.ttl␣friends.hdt
```

Subsequently, the second step involves generating family-based partitions by utilizing Command 3. To facilitate this process, we present a set of three distinct commands that have been executed to produce the settings employed in Example 4.3.1 (see p. 64), as follows:

Case Setting 1:

On using the native version:

```
./hdt-cpp-molecules/libhdt/tools/getFamilies
-s␣part\_friends-\␣-m␣100␣-e␣friends␣friends.hdt
```

Or on using the docker version:

```
docker␣run␣-v␣/host/path/target:/file/path/within/container␣--rm
getFamilies␣-s␣part\_friends-\␣-m␣100␣-e␣friends␣friends.hdt
```

Case Setting 2:

On using the native version:

```
./hdt-cpp-molecules/libhdt/tools/getFamilies
-s␣part\_friends\␣-L␣3␣-H␣20␣-c␣-e␣friends␣friends.hdt
```

Or on using the docker version:

```
docker␣run␣-v␣/host/path/target:/file/path/within/container␣--rm
getFamilies␣-s␣part\_friends\␣-L␣3␣-H␣20␣-c␣-e␣friends␣friends.hdt
```

Case Setting 3:

On using the native version:

```
./hdt-cpp-molecules/libhdt/tools/getFamilies
-s␣part\_friends\␣-L␣3␣-H␣20␣-m␣5␣-c␣-e␣friends␣friends.hdt
```

Or on using the docker version:

```
docker␣run␣-v␣/host/path/target:/file/path/within/container␣--rm
getFamilies␣-s␣part\_friends\␣-L␣3␣-H␣20␣-m␣5␣-c␣-e␣friends␣friends.hdt
```

[a] More details regarding these commands and functionalities can be found in the README document available at `https://github.com/smart-KG/smartKG-creator/tree/main/hdt-cpp-molecules/libhdt`

8.2 Deploying Our Experiments

In our experiment, we deployed and evaluated various web interfaces, including TPF, brTPF, SPF, SaGe, `smart-KG`, and `WiseKG`, using Docker containers. However, managing multiple containers manually can be a laborious task. To simplify this process, we utilized docker-compose, a tool that enables us to manage multiple containers at once through a YAML configuration file. With docker-compose, we were able to create and launch specific or all web interfaces with a single command.

As illustrated in Figure 8.1, our experimental setup[6] comprised of two virtual machines -

[6] More details regarding the experiments infrastructure including docker files, config files, scripts, and

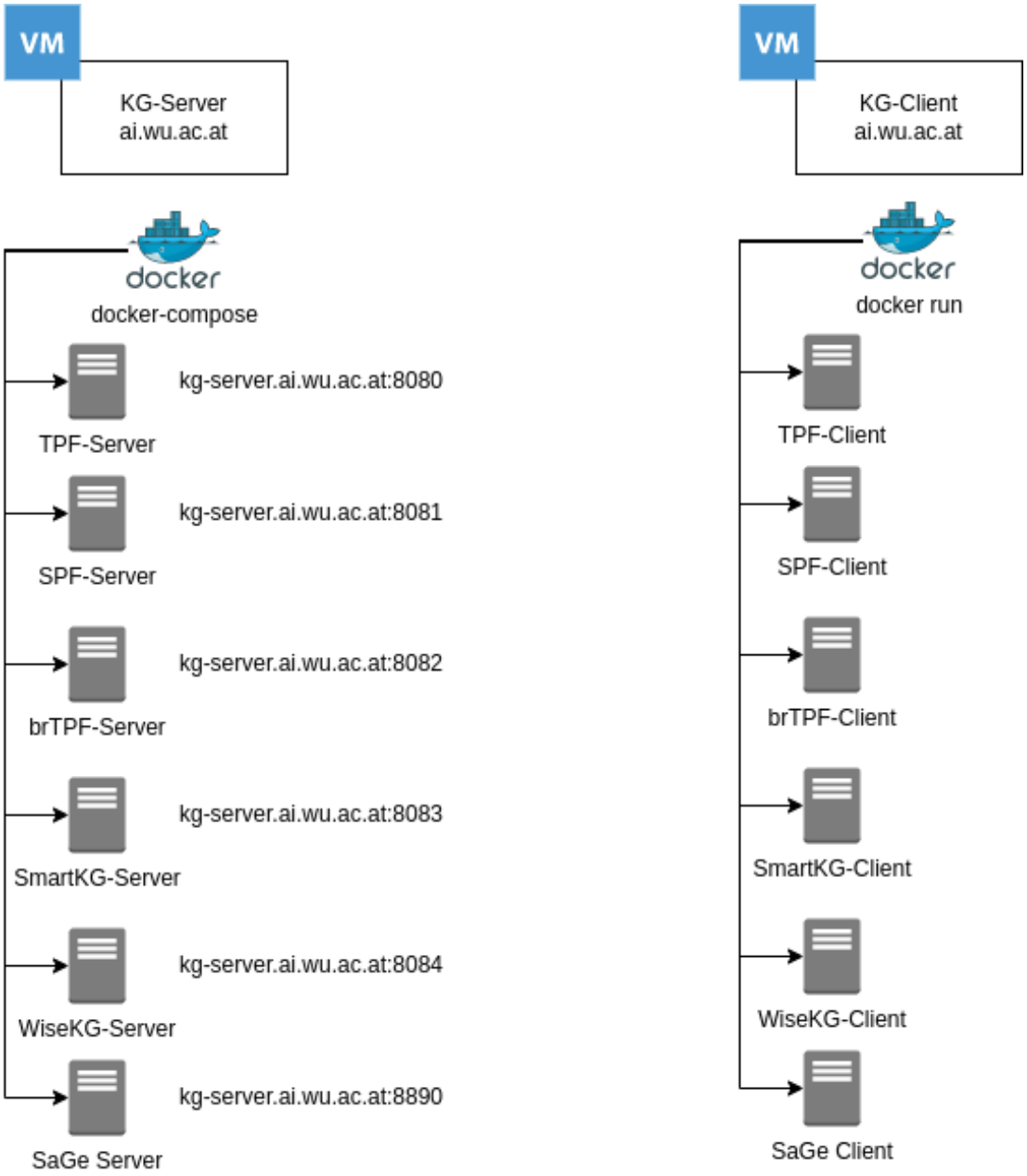


Figure 8.1: An illustration of our experiments infrastructure

one hosting the servers on the left-side and the other hosting the clients on the right-side. On the left side of the figure, we used docker-compose to create six containers, each corresponding to one of the compared systems' servers. On the right side of the figure, we demonstrated that by using Docker, we could instantiate one or multiple clients from

queries. https://github.com/AmrTAzzam/WebQuerying-Experiments/

each system simultaneously. For instance, TPF-Client(s) can communicate with the TPF-server container through the defined `http://kg-server.ai.wu.ac.at:8080/`.

We elaborate on how to build and deploy the docker image of our experiments in the following:

Command 4. *In the following, we write a command to start the docker decompose to run one server in our experiment, as follows:*

```
docker␣compose␣up␣[ContainerName]
docker␣compose␣up␣tpf-server
```

Here is the list of available containers: tpf-server, spf-server, brtpf-server, smartkg-server, wisekg-server, sage-server

Command 5. *We can also start all servers required for the experiments at once in the background, as follows:*

```
docker-compose␣up␣-d
```

Command 6. *The following command can be used to start one or multiple clients in the experiment setup by calling the docker container and assigning a single CPU for each container:*

```
docker␣run␣[arg]␣ldfclients␣[P]␣[filename].ext␣[S]␣[Q]␣[M]␣[O]␣[N]␣[ID]␣[D]
```

The docker image, ldfclients, contains all client implementations. The recommended arguments when running a docker container image are:

where we recommend the following set of arguments [arg] on running a docker container image [ldfclients] such as a java image:

- *-rm: This flag automatically removes the container when it exits.*
- *-cpus=1: This flag specifies the number of CPUs allowed for each container to utilize.*

In addition, each client should accept the following arguments:

- *[P] denotes the programming language required to execute the client which is Java in our Docker version of the clients. For java we use java -jar.*

- [*filename*].*ext*: *This argument denotes the executable file to run the client, such as smartkg-client.jar.*
- [*S*]: *This argument denotes the server API address to invoke a web querying interface, such as http://kg-server.ai.wu.ac.at:8083/.*
- [*Q*]: *This argument denotes the path of the directory to the query workload repository.*
- [*M*]: *This argument denotes the current method that is being evaluated, such as smart-KG.*
- [*O*]: *This argument denotes the output directory where the results of the execution, including all the evaluation metrics used in the experiments, are logged.*
- [*N*]: *This argument denotes the total number of concurrent clients in the current configuration.*
- [*ID*]: *This argument denotes the unique ID of the current client, which is used to access the correct query repository.*
- [*D*]: *This argument denotes the name of the dataset on which the experiment is being performed.*

  The command is eventually called from the server using a shell script, which is available at our repository[a]

[a] Shell Script to run instantiate the clients. https://github.com/AmrTAzzam/WebQuerying-Experiments/blob/main/dockerclients/run_experiments.sh

Example 11. *In the following, we detail an example of how to start an instance of* `WiseKG` *client and server for experiments:*

First, we begin with starting a `WiseKG` *server using the following command:*

```
docker␣compose␣up␣wisekg-server
```

This command will start a hosting URL for an RDF graph specified in the config file of each server for example dbpedia as follows: `kg-server.ai.wu.ac.at:8084/sparql/dbpedia`

Following, we instantiate the client(s) required for the experiments, as follows:

```
docker␣run␣--rm␣--cpus=1␣ldfclients␣-w="file/path/within/container"
-v␣/host/path/target:/file/path/within/container
```

```
openjdk:8␣java␣-jar␣wisekg-client.jar
http://kg-server.ai.wu.ac.at:8890/sparql/dbpedia
/test/queries/␣wisekg␣results_wisekg␣␣4␣4␣dbpedia2015
```

This command will start 4 wisekg clients and will concurrently run on the queries specified under the /test/queries.

8.3 Comunica Implementation

Apart from the above-described implementation, we also prototypically have developed another implementation based on Comunica [THSV18], a flexible research query engine for designing, developing, implementing, and evaluating both new and existing Web query interfaces, which helps improve extensibility and comparability.

As emphasized in this thesis, there is no silver bullet for Web querying, i.e., no particular interface works best in all settings, as there are tradeoffs inherent in each interface. We have obtained the best performance with `WiseKG`, which combines the strengths of several interfaces. This complicates, however, the client-side implementation, which, as in the case of `WiseKG`, must combine query results from heterogeneous interfaces.

To this end, we use Comunica [THSV18], a highly flexible framework that enables the evaluation of SPARQL queries on heterogeneous Web querying interfaces, including SPARQL endpoints, simple data dumps, TPF [VSH+16], brTPF [HA16], and `smart-KG` [AFA+20]. The main advantages of the Comunica meta query engine are the following:

- **Modularity:** Comunica provides an ideal research platform that allows customizing the query engine by inserting different components such as join algorithms, query planners and indexes, new or experimental SPARQL functions and to create new web interfaces, without code customization as in the case of Linked Data fragments implementation, but by integrating the needed modules through meta configuration. Thereby, we will be able to investigate cost models for `WiseKG` [AAM+21] in our future work.
- **Heterogeneity:** Comunica facilitates integration between heterogeneous Web query interfaces [ATP20b], which will accelerate the research process of hybridizing the multitude of existing LDF query interfaces; this is a result of relying on *Mediator Pattern*, which reduces the coupling of software components and consequently facilitates the exploration of a wide range of LDF approach combinations.

We have developed a Comunica package [ATP20a] for handling `smart-KG`. In the following, we detail the commands required to use `smart-KG` package[7] on top of Comunica:

[7] `smart-KG` package is built on Comunica 1.19. `https://github.com/comunica/comunica-feature-smartkg`.

Command 7. *The smart-KG package can be used by firstly cloning and installing Comunica which is requires requires Node.JS 8.0 or higher and the Yarn package manager as follows:*

```
git␣clone␣https://github.com/comunica/comunica.git

cd␣comunica

yarn␣install
```

Then, we install smart-KG package using the following command:

```
yarn␣add␣@comunica/actor-init-sparql-smartkg
```

To run a query on smart-KG, we can use this command:

```
comunica-sparql-smartkg␣[server]␣[query]

[server:]␣This␣argument␣is␣for␣the␣server␣URL.

[Query:]␣This␣argument␣takes␣a␣SPARQL␣query␣in␣a␣textual␣format.
```

In the following, we demonstrate the usage of `smart-KG` package in Comunica, as follows:

Example 12. *We provide an example that queries DBpedia endpoint to get the portrayers in Friends series using smart-KG package in Comunica:*

*comunica-sparql-smartkg http://fragments.dbpedia.org/2015-10/en "SELECT DISTINCT * WHERE dbo:friends dbo:starring ?portrayer."*

Also, we can start a SPARQL endpoint by providing a URL for the KG hosting server, for example:

```
comunica-sparql-smartkg-http␣http://fragments.dbpedia.org/2015-10/en
```

The endpoint is set to listen on port 3000 by default and can be accessed at `http://localhost:3000/sparql`. *Any client that supports the SPARQL protocol can send queries to this URL. The help command provides more details on the command:*

```
comunica-sparql-smartkg-http␣--help
```

As a future work, we plan to provide `WiseKG` package on top of Comunica v2.0.01 Our vision is to leverage Comunica as a key element for evaluating hybrid Web interfaces to guarantee the comparability and extensibility as well as the reproducibility of Web querying research.

CHAPTER 9

Conclusion

The number of open Knowledge Graphs on the Web has been growing steadily. The Linked Open Data community has been promoting the benefits of using RDF and Semantic Web technologies to standardize and interlink these graphs. As a result, there has been an increase in the amount of data published as Linked Open Data (LOD), forming a Web of interconnected Knowledge Graphs (KGs). This poses new challenges and research directions for scalable query processing on such RDF KGs published on the Web: querying large-scale KGs on the Web is difficult and costly, as full SPARQL query services are expensive to host and hard to maintain when dealing with complex and unpredictable query workloads and concurrent clients' requests. This precludes seamless live SPARQL querying of open KGs, which is the motivating premise behind this thesis.

To this end, we have proposed Web querying approaches that efficiently distribute the execution load of SPARQL queries between servers and clients. Our thesis presented a novel approach, Partition-based Linked Data Fragments interfaces, that involves combining existing LDF interfaces with shipping compressed graph partitions to the client-side that can be locally queried locally. Through this approach, we achieved a balance between server-side and client-side processing while improving the querying performance and reducing the average network traffic.

In Section 9.1, we summarize the primary contributions of this thesis. In Section 9.2, we critically assess our hypothesis and research questions. Finally, in Section 9.3, we provide an outlook on future directions and open challenges arising from this thesis.

9.1 Summary of Contributions

We now summarize our primary contributions to this thesis demonstrated in Chapters 3-6

Characterizing and Analyzing Web querying interfaces based on LDF

In Chapter 3, we began by borrowing from the original specifications of Linked Data Fragments to define, characterize, and analyze the existing LDF interfaces and our *proposed* partition-based LDF interfaces in a uniform view as instantiations of LDF. In this chapter, we align formal definitions and notations to uniformly present different hybrid interfaces that share the query processing load between clients and servers.

Leveraging Shipping Compressed KG partitions: Sharing the processing load

In Chapter 4, we presented our initial approach, `smart-KG`, for addressing the problem of SPARQL endpoints availability [VSH+16]. `smart-KG` shares the workload of query processing between the server and the clients by combining TPF and shipping compressed and queryable KG partitions that can be locally queried on the client-side.

We introduced a partitioning technique for RDF graphs called *family-based partitioning*, which groups entities with the same sets of predicates (i.e., families) into compressed, indexed, and queryable partitions. This approach is based on the observation that entities in RDF graphs are described using the same set of predicates and are often queried together.

In practice, the number of partitions generated by family-based partitioning can be large, making it prohibitively expensive to materialize all partitions for real-world RDF graphs. To address this issue, we introduced the concept of *predicate-restricted families*, which restricts the set of predicates considered for the partition creation process based on the predicates' cardinality.

To further optimize our partitioning technique to be utilized as a shipping strategy, we proposed a *family merging* strategy, which groups families with overlapping predicates into a single partition (see Example 4.3.1). This can serve to minimize the matching partition shipped to the client in response to a query. However, in practice, the number and size of partitions generated by merging all possible families in a graph can be too large to generate and ship over the network, especially for skewed RDF graphs. To address this issue, we proposed a *family pruning* strategy that restricts the number and size of materialized merged partitions.

To this end, `smart-KG` leverages these partitions to serve and ship only relevant partitions upon a query to be locally evaluated on the client-side.

Our experiments show that `smart-KG` has significantly outperformed the existing Web interfaces; in particular, (i) `smart-KG` only uses 60% of the CPU to handle the workload on 80 concurrent clients, increasing the availability of the server, while SaGe, for instance, exhausts the server CPU starting from 10 concurrent clients, (ii) `smart-KG` requires a significantly lower number of HTTP requests compared to TPF, reducing the network overhead, and (iii) finally, `smart-KG` reduces the overall shipped intermediate results for intensive workloads compared to TPF.

Employing a cost model: Dynamically shifting and balancing the load

In view of the findings from the previous chapter, we have identified the potential benefit of dynamically deciding whether a subquery should be performed on the client or the

server. Our observations show that (i) shipping KG partitions has reduced the server workload, but there is still room to better utilize server resources while maintaining high performance, especially during high query processing load, (ii) the characteristics of subqueries, such as selectivity and intermediate result cardinality estimation, have a major influence on the execution location, and (iii) several parameters, including client and server resource availability and estimated network time, determine where to process a certain part of a query.

Based on these observations, in Chapter 5 we described our system `WiseKG`, which dynamically delegates query processing based on the current server workload and client capabilities. Leveraging two complementary interfaces, `smart-KG`, which executes star-shaped subqueries on the client-side and SPF, which executes star-shaped subqueries on the server-side in two different ways: (i) using a heuristic approach, where we perform all queries on the server up to a certain threshold, and then switch to the client-side until the CPU load falls back below the threshold, and (ii) using a cost model, which considers parameters such as CPU load, estimated network transfer time, and currently available resources at the client to dynamically pick the execution location.

In our extensive evaluation, we found that the heuristic version of `WiseKG` already performed better than all state-of-the-art interfaces in terms of scalability and performance. On top of that, The cost model version further outperformed the heuristic one. `WiseKG` is up to four times faster than SPF and `smart-KG`, and up to an order of magnitude faster than other existing LDF interfaces such as SaGe and TPF over watdiv1B with a load of 128 concurrent clients. When queries were performed over the watdiv1B dataset with a 5-minute timeout threshold, WiseKG only incurred 2% timeouts of the total workload queries, compared to the percentages of timeouts that reached 13%, 21%, 55%, and 56% for `smart-KG`, SPF, SaGe, and TPF, respectively.

Partition-Based Linked Data Fragments: refinements

In Chapter 6, we investigated possible concrete interfaces that can ship partitions of KGs from the server to the client in order to reduce server-resource consumption. Motivated by the previous findings, which showed the beneficial effects of family partitioning on the scalability and performance of Web interfaces, we introduced the concept of Partition-Based Linked Data Fragments. These approaches can be viewed as a generalization of existing LDF interfaces, which ship compressed and queryable partitions that can answer admissible query patterns (e.g. a star or a single triple pattern) on the client-side with minimal server interactions, aiming to reduce data transfer, the number of requests and increase server availability.

As part of our investigation, we presented a formalization of existing KG partitioning techniques, such as horizontal, vertical, and others, in order to uniformly discuss their applicability to serve as a partitioning shipping mechanism for partition-based interfaces under the umbrella of the LDF framework.

Based on this generalization, we presented a concrete implementation of a partition-based LDF approach, called `smart-KG`$^+$. Our approach uses an extension of family

partitioning called typed family-partitioning, which exploits horizontal partitioning to subdivide families based on the classes of the entities belonging to each family.

We have conducted an extensive evaluation of smart-KG+, which consists of two parts: first, we evaluate a set of implementation refinements, including the use of asynchronous join to produce results in an incremental fashion, and a server-side query planner. Second, we evaluate the effectiveness of typed-family partitioning, which shows that it requires up to 41% and 46% less transferred data over watdiv10M and watdiv100M on intensive workloads. Moreover, when considering star queries with the "rdf:type predicate", a widely prevalent pattern in query logs of current SPARQL endpoints, typed-family partitioning requires fewer data transfers than family partitioning.

In Chapter 7, we reviewed and briefly discussed how or whether other graph partitioning techniques from the literature could be used in our framework.

Finally, In Chapter 8, we present two openly available implementations that shall allow other researchers to modularly represent and extend our results.

9.2 Critical Assessment of Research Questions

In light of what we have summarized thus far, we critically review our research questions associated with the hypotheses originally introduced in Chapter 1,

RQ1 *Can we achieve speedups to existing Web querying approaches over Knowledge Graphs by leveraging partition shipping?*

The answer is yes, we provided a detailed answer for this question in Chapter 4 where we introduced smart-KG, a novel hybrid strategy for improving the performance of Web querying over Knowledge Graphs by achieving a more balanced client-server load distribution. smart-KG combines TPF with the shipping of compressed KG partitions and differs from TPF in that it ships modular, query-relevant KG partitions that can be directly queried by a client, rather than relying solely on shipping intermediate results for a specific triple pattern.

Our empirical evaluation demonstrates that smart-KG improves server availability while maintaining superior query performance compared to the state-of-the-art approaches, particularly with an increasing number of clients and increasing KG sizes, with the fastest average workload completion time per client and the least number of query timeouts. This is achieved through partition shipping and triple pattern lookup, which are low-CPU operations that reduce the overall CPU usage of the server. As a result, the server can handle a higher number of requests without becoming overloaded. In addition, smart-KG significantly reduces the average number of requests sent to the server compared to other approaches and requires fewer data transfers than TPF, even up to 10 times less, but more data transfer than SaGe. These benefits are demonstrated in experiments and discussed in more detail in Chapter 4.

In Chapter 4, we also demonstrate the feasibility of this family-based partitioning as a shipping strategy through experiments using the synthetic WatDiv dataset and real-world datasets such as DBpedia, dblp, WordNet, and Yago2. Our experiments show that, for example, the 1-billion triple WatDiv graph took 12 hours and 300GB of disk storage space to materialize 52,885 partitions, while DBpedia, with a size of 837M triples, took 23 hours and 122GB of disk storage space to generate 29,965 partitions. The retrieved partitions in `smart-KG` can be reused for queries that require the same partitions, and we demonstrate that caching shipped partitions can be useful for executing streak queries with minimal communication to the server.

We argue that family-based partitioning offers a reasonable trade-off between query execution time and scalability. Our experimental evaluation shows that it has sub-linear computational time, with the graph size determining the time required to generate the partitions. It also has a storage space cost that is estimated to be about double the raw size of the WatDiv graphs (DBpedia uses even less space due to more restricted pruning we apply). This cost is manageable given the availability and affordability of disk space, and HDT partitions can be further compressed using lossless techniques such as gzip[1], and 7-Zip[2]. Overall, our results suggest that family-based partitioning is an effective and scalable approach to support Web querying large-scale RDF graphs. We even further improved these speedups with the extensions of our approach presented in Chapters 5 and 6.

RQ2 *How can the combination of different LDF interfaces impact the query performance?*

To address research question RQ2, we present two possible strategies in Chapter 5 that combine the strengths of state-of-the-art interfaces such as brTPF, SPF, and `smart-KG`. These strategies are: (i) a simple *Heuristic Model*, which is based on a predefined CPU usage threshold to dynamically switch between server-side and client-side star subquery evaluation, and (ii) a *Cost Model*, which is inspired by the classic R^* algorithm from distributed databases. We provide an extensive evaluation of both strategies and found that combining `smart-KG` and SPF has significantly enhanced the workload completion time.

In Chapter 5, we further provide more details on how to effectively and efficiently combine different interfaces to improve query performance and server scalability. Different interfaces have varying levels of expressivity, which affects the amount of client and server processing and data transfer required for a given query. For instance, `smart-KG` may require less server processing but more client processing, while SPF may require more server processing but less client processing. By combining both interfaces and selecting the most appropriate interface for a given query based on its characteristics and the available resources on the client and server. In conclusion, we improved the overall query performance while providing a

[1] `https://www.gzip.org/`
[2] `https://www.7-zip.org/`

combined client-server Web querying interface. By additional optimizations and a redesign of `smart-KG` component presented in Chapter 6, we could even further improve the performance of the combined approach.

RQ3 *How can we further refine and optimize the partitioning technique?*

In Chapter 6, we provide further optimization to the shipping strategy introduced in Chapter 4. First, we introduced `smart-KG`$^+$ which utilizes typed-family partitioning as a shipping strategy. In addition, `smart-KG`$^+$ benefits from the accurate plans generated by our server-side shipping-based query planner and optimizer. Lastly, `smart-KG`$^+$ can asynchronously join intermediate results from heterogeneous LDF interfaces to improve query performance. The improved version, called `smart-KG`$^+$, resulted in faster workload completion times, fewer timeouts, and higher throughput compared to other systems, such as `smart-KG`, TPF, and SaGe, when tested with the `watdiv-sts` workload on RDF knowledge graphs of various sizes. The improvement in performance is due to server-side query planning and the use of brTPF, which reduces the number of HTTP requests. `smart-KG`$^+$ performs particularly well with a high number of concurrent clients, with up to 18 and 7 times faster workload completion times than TPF and SaGe, respectively, for a workload with a single client, and up to 3 and 2.6 times faster for a workload with 128 concurrent clients. `smart-KG`$^+$ also had a lower percentage of timeouts than TPF and SaGe, especially when the RDF knowledge graph size was large. In terms of throughput, `smart-KG`$^+$ had higher values than the other systems, reaching up to 4132, 678, and 109 queries per minute for 1, 32, and 128 concurrent clients, respectively, on the `watdiv1B` dataset.

In Chapter 7, we also analyze other existing graph partitioning techniques and their potential use as a shipping strategy in the context of Web querying. We focus on techniques such as vertical partitioning, hash partitioning, and horizontal partitioning. Through our analysis, we conclude that some of these techniques have the potential to serve as effective partitioning mechanisms for Web querying, but further experimentation is needed to fully evaluate their performance and effectiveness. We, therefore, leave the experimental evaluation of these techniques as a topic for future work. In order to facilitate further investigation into shipping strategies, in Chapter 8, we provide our software in a reproducible fashion through two distinct implementations, one based on extending the implementation of the Linked Data Fragment framework Comunica and the other based on Comunica framework.

9.3 Open Challenges and Future Research Directions

In this thesis, we have demonstrated that efficient and scalable SPARQL querying is feasible over large-scale Knowledge Graphs (KGs) on the Web in the order of a billion triples up to hundreds of concurrent clients. We now focus on promising and worthwhile future directions to explore - from our perspective - arising from the work presented in

this thesis. In particular, we describe several open challenges that we plan to address in our future work:

- **C1 Querying of Evolving Knowledge Graphs:** Our current partitioning strategy does not provide efficient update strategies, which poses a significant obstacle in serving dynamically evolving KGs in real-time. For instance, the generation of the required partitions for large-scale KGs such as DBpedia can take up to 23 hours, making it challenging to keep up with the dynamic evolution of the data.
- **C2 Cardinality Mis-Estimations:** In our thesis, we propose a cost model which leverages characteristic sets as a cardinality estimation to generate query execution plans. However, characteristic sets rely on the assumption of independence between star-subqueries in a given query. In cases where these subqueries are correlated, this assumption may not hold and can result in a significant underestimation of the selectivity of the query especially when the result size increases, leading to inaccurate cardinality estimates [PKB+20].
- **C3 Conflicting Optimization Objectives:** So far, we have primarily adopted a classical query optimizer which estimates the cost of a query plan as a scalar cost value, representing the estimated total execution time of a given query. While – as we have experimentally demonstrated – the cost model is very effective, it overlooks the following issues: (i) it only focuses on only one objective which is minimizing the query execution time, however, there are several scenarios in which multiple other (often conflicting) objectives have to be jointly considered during the query optimization process, and (ii) it observes the system parameters (e.g. CPU usage, network, and disk bandwidth) as a momentary snapshot (over the last minute). Strictly speaking, in our current cost model, the relevant values of the system parameters are known at run time, not at the query plan compilation time. Hence, our execution plan for a given query could potentially be sub-optimal whenever the prior assumed system's values at planning time mismatch the actual values at run time.
- **C4 Limited Resources of Client-Server Architecture:** While our introduced Web interfaces offer scalable and reliable Web querying to large KGs, LDF interfaces still strictly follow client-server architecture which could potentially lead to network congestion, downtimes, limited processing resources, and high costs to host the servers. Besides, our proposed solutions give a fair bit of responsibility to data providers to maintain access to the Web interfaces. For instance, as we earlier discussed, our proposed family-based partitioning and cost model requires *parameter tuning* to provide high-performance Web querying services, adding an extra burden on data providers.
- **C5 High Preprocessing Overhead** Graph partitioning techniques have been shown to enhance query performance in various querying environments. However,

the use of such techniques can result in significant preprocessing overhead, as they may need to be applied to the entire dataset, or certain thresholds may need to be set to regulate the number and size of partitions generated. In real-world scenarios, it is typically the case that only a small proportion of the data is accessed. This has been demonstrated in empirical studies such as [RHSG14], and likely also applies to our pattitions.

- **C6 Limited Real-world Web Querying Benchmarks** At present, RDF data management research mostly relies on synthetic benchmarking frameworks for SPARQL querying performance evaluation, including Waterloo SPARQL Diversity Test Suite (WatDiv) [AHÖD14], LargeRDFBench [SHN18], the Berlin SPARQL Benchmark (BSBM) [BS09], SP2Bench [SHLP09] which are generated based on uniform and structured schemas that are often less complex than organic and community-driven ones such as Wikidata and DBpedia. Thus, we believe that such benchmarks do not accurately simulate the skewed and semi-structured nature of real-world RDF KGs, and results obtained using such synthetic RDF KGs in terms of scalability and performance do not necessarily reflect those that would be obtained in case of real-world KGs [DKSU11]. In this thesis, while we use both synthetic and real-world KGs for our experimental evaluation, we use a small number of hand-selected DBpedia queries from real-world query logs either extracted from LSQ query log [SAH+15] or generated from FEASIBLE benchmark [SMN15]. In general, real-world benchmarks such as FEASIBLE [SMN15] and DBpedia SPARQL Benchmark (DBSBM) [MLAN11] focus on generating query templates with different characteristics. However, none of the currently existing real-world benchmarks offer diversified stress testing query workloads which play a vital role in the evaluation of Web querying interfaces.

In order to tackle the aforementioned challenges, we propose potential future directions which can be complementary and interconnected, as follows:

- designing *KG partitions update strategies* to enable KG querying over continuously evolving KGs;
- investigating cardinality estimations techniques from the literature to provide more accurate cost estimations;
- investigating the server availability and the query performance trade-off as *multi-objective query optimization* or *multi-objective parametric query optimization* problem, taking multiple query execution cost metrics into account;
- considering the migration of Web Querying to *collaborative decentralized architecture* to lift the burden of managing the query service off the data providers;
- investigating *workload-based partitioning* to cover the most frequent queries in the logs;

- exploring *machine learning approaches* for Web querying optimization, leveraging the ever-increasing query logs on the available SPARQL endpoints;
- providing an empirical evaluation for the other potential *partition-based LDF interfaces* introduced in Chapter 7;
- designing and creating an *evaluation framework* for Web querying based on real-world KGs.

Efficient partitions update strategies

In this thesis, we have demonstrated the feasibility of the family partitioning generation process. However, we did not consider the scenario of data dynamicity: our partitioning technique assumes that the input is a static snapshot of an RDF KG and will remain static during the analysis time. To tackle challenge **C1**, we envisage proposing efficient partitioning strategies whereby the updates are continuously stored on a separate data storage offering live updated query results in combination with the existing partitions. In the meantime, we locate the affected partitions with the updates and regenerate them, preventing the full regeneration process. Still, we could imagine that many of data consumer applications potentially demand Web querying for continuously evolving KGs which is not attained by our current assumption of static KGs. However, we still believe that our proposed solution is applicable and, for such a scenario, subject to further research; for example, assuming that the majority of data publishers provide (almost) static KGs, our Web querying interfaces could potentially rely on our partition generator for the slow-evolving portions, and handle dynamic portions with HBase or PostgreSQL similar to SaGe [MSM19]. Still, while the partition generation for a large-scale RDF graph such as DBpedia requires half a day, the applicability of our work for more dynamic evolving KGs as well as stream processing is an open research question.

Exploring the literature of cardinality estimation

Cardinality estimation is a crucial element of finding a balanced query processing for the Web querying interfaces. We plan to investigate diverse cardinality estimators in order to address challenge **C2**. As reported in [LGM+15], in the context of the relational database, the cardinality misestimates dominate the query execution time compared to the cost model errors.

There is a body of existing research work towards a better cardinality estimation of result sizes of SPARQL queries based on sampling-based techniques such as IMPR [CL18] and graph summarization-based techniques for instance characteristic sets [NM11] and SumRDF [SMK18]. It would be an interesting research direction if we explore the aforementioned techniques and adopt them in future Web querying systems.

G-CARE [PKB+20] is a novel framework that has recently investigated the performance of several cardinality estimation techniques for subgraph matching including techniques designed to estimate RDF data such as [CL18, SMK18, NM11] in addition

to techniques selected from the context of a relational database, for instance, CorrelatedSampling [VMZC15] and WanderJoin [LWYZ19]. The experiments reported that WanderJoin has outperformed the aforementioned techniques in q-error which is the cardinality estimation accuracy measure. G-CARE has conducted an evaluation on real-world datasets such as LUBM, YAGO, and DBpedia on different query topologies such as chains, stars, trees, and cycles.

This thesis utilizes characteristic sets as the cardinality estimation function $CE(.)$ in `WiseKG`, as it is a pre-processing step in generating family partitions (i.e., family generator). This enables us to leverage the storage requirements and summary structure built during family generation. However, to fully assess the effectiveness of using characteristic sets as compared to alternative cardinality estimators on the overall performance of our systems, further experimental evaluations are necessary.

Exploring multi-objective (parametric) query optimization

To address challenge **C3**, an interesting research direction would be to consider multi-objective (parametric) query optimization inspired by the database literature. Multi-objective query optimization [TK14, HLY93, KSTI11, BG04] includes creating a Pareto-optimal query plan, providing the best possible compromise according to conflicting objectives under multiple cost metrics. In our current Web querying optimization scenario, with the aim to serve multiple users concurrently, multi-objective query optimization could be a potential solution to optimize two conflicting objectives: minimizing the server resources consumption (i.e. increasing server availability) while minimizing the query execution time. Another scenario is that both data publishers and consumers have conflicting objectives, namely minimizing the query execution time and minimizing the monetary costs (i.e. the expense of local server computation, the charge of transferring data or intermediate results to the clients for further querying).

Next, we believe that multi-objective parametric query optimization [BBD09, HS02, Gan98, TK17] could be a promising research direction to address the issue (ii). In essence, multi-objective parametric query optimization provides multiple candidate plans for a given query, each plan is an optimal plan for a combination of parameter values that are unknown in the compilation time. At run-time, the system invokes a query plan that is optimal for the actual parameter values. At a prepossessing step, for each query template, candidate query plans will be calculated and stored and eventually utilized to provide an optimal query based on the actual values during the processing time. In addition, parametric query optimization could potentially give the query consumer the freedom of selecting a reasonable trade-off according to their needs.

Exploring collaborative decentralized architectures

Looking towards the future, we spot a great need for more decentralized architecture for available, queryable, and up-to-date Web knowledge graphs. This envisioned requirement is hard to achieve in reality by many data providers. Hence, we plan to explore decentralized peer-to-peer architectures for publishing and querying KGs. We believe that each

participating node could potentially provide client and server functionality with respect to KG partitions generation, replication, and querying to address challenge **C4**.

Recent studies [KSR+07, KKK+10, CF04, AMH19b, AMH19a] have proposed decentralized peer-to-peer architecture for efficient query processing over RDF KGs. However, the current decentralized systems suffer from significant communication overhead, inaccurate query planning, and imbalanced query processing load distribution leading to a low querying performance. Therefore, we believe that our findings in this thesis could potentially improve and further optimize collaborative decentralized query processing, as follows (i) we plan to explore a replication and fragmentation technique for KGs between the nodes based on family partitioning, (ii) we aim to not only decentralize the query processing but also the partitioning generation process, contrast to the current proposal where data publisher is fully responsible for the partition generation, (iii) we aspire to devise query plans that are aware of data fragmentation and placements, network peer capabilities, and the dynamic adjustment of the network.

To this end, we plan to explore the migration of Web querying from the resource limitations of the client-server architecture to a collaborative decentralized architecture managed, stored, and queried by the knowledge graph community users.

Investigating workload-aware partitioning

As a future direction, we propose to investigate the use of workload-based partitioning [ASY+21] for Web querying interfaces to tackle challenge **C5**. While this approach has been applied to distributed and cloud RDF graph processing systems such as WARP [HS13], Partout [GHS14], WORQ [MAA18], and AdPart [AAK+16], it has not yet been studied in the context of Web querying.

To avoid overfitting, it is important to use a representative RDF query log that accurately reflects the usage patterns of the data [HS13]. By applying workload-based partitioning to Web querying interfaces, we aim to improve the performance and scalability of RDF Web KG querying while minimizing preprocessing overhead and ensuring that the partitions accurately match the access patterns of the data users. This will involve using query logs to identify common joins and partition or replicate parts of the graph, and applying the partitioning a priori or dynamically as queries are received [ASY+21]. We plan to utilize query logs to create partitions that accurately match the access patterns of the specific RDF graph, allowing for more accurate query planning and optimization.

Exploring *machine learning approaches* for Web querying optimization

Machine learning approaches have been widely used in relational database systems to enhance their performance and scalability [WZC+16]. Our hypothesis is that the utilization of machine learning techniques has the potential to address the challenges **C2-C5**. For instance, we can leverage ML techniques to predict query execution times, more accurate cardinality estimation, and predict query workloads which will enable us to enhance the query performance by better utilizing the available resources.

By analyzing query logs and identifying common access patterns, these systems can generate partitions [YYG19, KDZ+17], physical designs [PAA+17], join orderings [MP18, KYG+18], cost estimations [SL19] and indexes [KBC+18, DMY+20] that match the needs of the users, allowing for more efficient query processing and optimization [MP19b, MNM+19]. Additionally, machine learning algorithms can be used to develop dynamic, adaptable optimization strategies [TWW+21, OBGK18] that can continually update and adjust to changing usage patterns [PAA+17, MP19a]. This can enable the system to automatically adjust its performance and resource allocation [ZLZ+19, LZLG19] based on changing concurrent workloads [DÇPU11, ZSLF20, WCHN13] and usage patterns [MAH+18, YLCL21, LZLG19], resulting in improved efficiency and scalability.

Inspired by the role of machine learning in relational database systems, SPARQL Web querying interfaces can potentially benefit from the application of machine learning techniques. By analyzing past query performance data from query logs, a SPARQL Web querying interface can learn which query execution plans are most effective for a given dataset and hardware configuration, enabling the system to more efficiently identify the optimal query execution plan [KYG+18], which can potentially improve overall performance. For example, using machine learning to consider available indexes (such as HDT indexes) and the RDF data distribution in the knowledge graph can help to improve cost estimations. Additionally, machine learning can be used to automatically tune system parameters, such as those used in a family partitioning mechanism, by suggesting suitable parameters based on query logs simplifying the process of data management for data publishers. By continuously learning from the SPARQL query logs, the system can automatically adjust these parameters to ensure efficient query execution, especially for large RDF knowledge graphs scenarios where choosing appropriate parameters can be complex and time-consuming for the data publishers. Another potential benefit of using machine learning for Web interfaces is improved reliability and uptime. Machine learning algorithms can be used to monitor the health and performance of the Web interface, and automatically take corrective action if any issues are detected during high concurrent query workload (for example, by switching to a less expressive interface or pushing the workload to the client-side [DÇPU11, ZSLF20]).

Overall, the use of machine learning in SPARQL web querying interfaces can help to improve the performance and availability of these systems, making them more effective at handling complex and large RDF KGs.

Empirical evaluation for other partition-based LDF interfaces

In Chapter 7, we proposed alternative partition-based LDF interfaces based on several existing partitioning techniques. We plan to conduct an empirical evaluation of these interfaces to assess their performance and effectiveness using real-world RDF knowledge graphs and standardized metrics, similar to experiments done with family partitioning. This empirical evaluation can identify potential data-transfer improvements and query optimization opportunities, as well as potential partitioning techniques that can enhance Web querying availability, based on particular structures and data distributions in KGs.

Evaluation framework for Web querying

Looking towards the (near) future, we thus see a great need for designing and creating stress-testing benchmarking frameworks in order to tackle challenge **C6**, based on real-world KGs and queries, to generate diverse and intensive query workloads in order to evaluate the scalability of Web querying interfaces under multiple concurrent clients such as ours. Such frameworks will empower researchers to produce relevant and high-quality research ready for real-world Web querying challenges based on a stress testing process and also prevent the hustle of query sampling, manual query selection, and subjective (non-standard) evaluation. More ambitiously, although very challenging, we believe that a suitable stress testing framework over real-world KGs could potentially minimize the dependence on synthetic benchmarks fulfilling the current requirements of Web-scale querying interfaces.

List of Figures

2.1 A graphical representation of an RDF graph from Examples 2 and 3. The nodes represent subjects and objects and the directed labeled edges represent the predicates. 18
2.2 The LOD cloud diagram, as of October 2022 21
2.3 A left-linear execution plan based on Nested Loop Join based on a greedy heuristic 28
2.4 Examples of different query plans of a given SPARQL query 29

4.1 KG example 52
4.2 Predicate families for the KG shown in Fig. 4.1 52
4.3 Overall architecture for the `smart-KG` client and server. 53
4.4 Example of processing a SPARQL query with the `smart-KG` client. . . . 61
4.5 The query plan of Example 4.4a according to TPF implementation that we have followed in our Experiments 73
4.6 Ablation study in DBpedia to select the parameters in partitioning algorithm 75
4.7 Performance on the WatDiv-100M workload 76
4.8 Performance on the workloads (80 clients) at increasing KG sizes 77
4.9 Average execution time (80 clients) with DBpedia high-demanding queries 78
4.10 Avg. execution time per client on the standard WatDiv-100M for L queries 80
4.11 Avg. execution time per client on the standard WatDiv-100M for S queries 81
4.12 Avg. execution time per client on the standard WatDiv-100M for F queries 82
4.13 Avg. execution time per client on the standard WatDiv-100M for C queries 83
4.14 Complex Queries Workload 83
4.15 Server-resources consumption on the intensive workload. 86
4.16 Client CPU and RAM usage on the intensive workload at increasing sizes 87

5.1 Example of processing a SPARQL query with `WiseKG` 91
5.2 Number of timeouts, average workload time, and throughput for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` on `watdiv-sts` 102
5.3 Execution time (in seconds) for 28 diverse queries over the `dbpedia` dataset. 104
5.4 Avg. execution time per client over `watdiv100M` for the `watdiv-btt` workload. 105
5.5 Impact of the cost model components on the performance and resources consumption over `watdiv100M` 106

5.6 Number of requests to the server and number of transferred bytes for 128 clients over watdiv10M, watdiv100M, and watdiv1B, and CPU load for increasing numbers of clients over watdiv1B on the watdiv-sts workload 108
5.7 Avg. Server CPU Usage (in %) for increasing numbers of clients over watdiv1B. 108

6.1 KG example 116
6.2 Predicate families and typed families for the KG shown in Fig. 6.1 116
6.3 The overall architecture of the smart-KG$^+$ client and server, wherein the modified components are denoted in red, in contrast to the corresponding elements in the original smart-KG architecture. 117
6.4 Example of processing a SPARQL query with the smart-KG$^+$ client. . . 120
6.5 Workload completion time (lower is better) 141
6.6 Number of timeouts (lower is better) 142
6.7 Query throughput of 128 concurrent clients over watdiv10M, watdiv100M, and watdiv1B datasets on watdiv-sts workload 143
6.8 Query throughput for increasing numbers of clients over watdiv1B on watdiv-sts workload 143
6.9 Throughput (higher is better) 143
6.10 Average first result time for 128 clients over increasing sizes datasets on watdiv-sts workload 144
6.11 Average first result time for increasing number of clients over watdiv1B dataset on watdiv-sts workload 144
6.12 Average first result time (lower is better) 144
6.13 Avg. execution time per client on WatDiv-100M, for the first query of each category L, S, F, and C 146
6.14 Avg. execution time per client on the standard WatDiv-100M, for simplest L queries 149
6.15 Avg. execution time per client on the standard WatDiv-100M, for Star S queries 150
6.16 Avg. execution time per client on the standard WatDiv-100M, for Snowflake F queries 151
6.17 Avg. execution time per client on the standard WatDiv-100M, for Complex C queries 151
6.18 Server resource consumption with increasing number of clients and increasing dataset sizes on watdiv-sts workload 153
6.19 Data transferred per query (in bytes) over watdiv10M and watdiv100M on watdiv-btf$_{unbounded}$ and watdiv-btf$_{bounded}$ workloads 156
6.20 Execution time per query (in ms) over watdiv10M and watdiv100M on watdiv-btf$_{unbounded}$ and watdiv-btf$_{bounded}$ workloads 157
6.21 Execution time per query (in sec) on DBpedia-btt$_{bounded}$ workload . . . 158

8.1 An illustration of our experiments infrastructure 177

List of Tables

3.1 Aligned formal definitions and notations with LDF original specifications to uniformly present different existing LDF APIs 42

4.1 Characteristics of the evaluated knowledge graphs 72
4.2 Comparison of storage requirements (in MB) for systems with HDT backend vs original graph size (raw) 85

6.1 Characteristics of the evaluated knowledge graphs 134
6.2 Evaluation Workloads Statistics. We provide the total numbers for all the 128 clients 136
6.3 Family-based Partitions Parameter Settings 138
6.4 Characteristics of additional real-world knowledge graphs 139
6.5 An ablation study to assess the performance of each individual contribution over `watdiv10M` using `watdiv-btt` workload. (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in ms, TO: Timeouts). GM-T = Total Geometric mean for all query classes 139
6.6 An ablation study to assess the performance of each individual contribution over `watdiv10M` using `watdiv-sts` workload. (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in ms, TO: Timeouts). GM-T = Total Geometric mean for all query classes 140
6.7 Avg. execution time per client (in sec.) for 128 clients over `watdiv100M` for the `watdiv-btt` workload. GM=Geometric Mean per query class. GM-T = Total Geometric mean for all query classes. 147
6.8 Comparison of storage requirements (in MB) for systems with HDT backend vs original graph size (raw) 154
6.9 `Workload Transferred Data` per client over `watdiv10M` and `watdiv100M` on $\texttt{watdiv-stf}_{bounded}$, $\texttt{watdiv-stf}_{unbounded}$ and $\texttt{watdiv-stf}_{both}$ workloads 155
6.10 `Workload Completion Time` per client over `watdiv10M` and `watdiv100M` on $\texttt{watdiv-stf}_{bounded}$, $\texttt{watdiv-stf}_{unbounded}$ and $\texttt{watdiv-stf}_{both}$ workloads 155
6.11 Impact of Typed-Family Partitioning on `WiseKG`'s Performance on `watdiv10M` dataset (Req: Requests, DT: Data Transfer in MB, ET: Execution Time in milliseconds, TO: Timeouts. ET is in milliseconds, DT is in MB 159

7.1 An overview of the exiting graph partitioning techniques utilized in RDF engines 164

List of Algorithms

4.1 Family Grouping 57
4.2 Query Executor: *evalPlan* 63

5.1 Create an annotated query execution plan 97
5.2 Processing a Query Execution Plan 98

6.1 Query Optimizer and Planner: *optimizePlan* 123
6.2 Query Executor: *evalPlan* 125
6.3 Query Executor: $eval_c$ 126

Bibliography

[A+99] Werner Almesberger et al. Linux network traffic control implementation overview, 1999.

[AAA+15] Alexander Aarts, Joanna Anderson, Christopher Anderson, Peter Attridge, Angela Attwood, Jordan Axt, Molly Babel, Štěpán Bahník, Erica Baranski, Michael Barnett-Cowan, Elizabeth Bartmess, Jennifer Beer, Raoul Bell, Heather Bentley, Leah Beyan, Grace Binion, Denny Borsboom, Annick Bosch, Frank Bosco, and Mike Penuliar. Estimating the reproducibility of psychological science. *Science*, 349(6251):aac4716, 2015.

[AAC+19] Amr Azzam, Peb Ruswono Aryan, Alessio Cecconi, Claudio Di Ciccio, Fajar J. Ekaputra, Javier D. Fernández, Sotiris Karampatakis, Elmar Kiesling, Angelika Musil, Marta Sabou, Pujan Shadlau, and Thomas Thurner. The cityspin platform: A CPSS environment for city-wide infrastructures. In Antonella Longo, Maria Fazio, Rajiv Ranjan, and Marco Zappatore, editors, *Proceedings of the 1st Workshop on Cyber-Physical Social Systems co-located with the 9th International Conference on the Internet of Things (IoT 2019), Bilbao, Spain, October 22, 2019*, volume 2530 of *CEUR Workshop Proceedings*, pages 57–64. CEUR-WS.org, 2019.

[AAK+16] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB J.*, 25(3):355–380, 2016.

[AAM+21] Amr Azzam, Christian Aebeloe, Gabriela Montoya, Ilkcan Keles, Axel Polleres, and Katja Hose. Wisekg: Balanced access to web knowledge graphs. In Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia, editors, *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pages 1422–1434. ACM / IW3C2, 2021.

[ÁBFM11] Sandra Álvarez-García, Nieves R. Brisaboa, Javier D. Fernández, and Miguel A. Martínez-Prieto. Compressed k2-triples for full-in-memory RDF engines. In Vallabh Sambamurthy and Mohan Tanniru, editors, *A Renaissance of Information Technology for Sustainability and Global Competitiveness. 17th Americas Conference on Information Systems, AMCIS 2011, Detroit, Michigan, USA, August 4-8 2011*. Association for Information Systems, 2011.

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

[ACHZ09] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing linked datasets. In Christian Bizer, Tom Heath, Tim Berners-Lee, and Kingsley Idehen, editors, *Proceedings of the WWW2009 Workshop on Linked Data on the Web, LDOW 2009, Madrid, Spain, April 20, 2009*, volume 538 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[AFA+20] Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. SMART-KG: hybrid shipping for SPARQL querying on the web. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 984–994. ACM / IW3C2, 2020.

[AHKK17] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *Proc. VLDB Endow.*, 10(13):2049–2060, 2017.

[AHÖD14] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2014.

[AHUV13] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL web-querying infrastructure: Ready for action?

In Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, volume 8219 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2013.

[AKMH20] Christian Aebeloe, Ilkcan Keles, Gabriela Montoya, and Katja Hose. Star pattern fragments: Accessing knowledge graphs through star patterns. *CoRR*, abs/2002.09172, 2020.

[AKP18] Amr Azzam, Sabrina Kirrane, and Axel Polleres. Towards making distributed RDF processing flinker. In *4th International Conference on Big Data Innovations and Applications, Innovate-Data 2018, Barcelona, Spain, August 6-8, 2018*, pages 9–16. IEEE, 2018.

[AMH19a] Christian Aebeloe, Gabriela Montoya, and Katja Hose. A decentralized architecture for sharing and querying semantic data. In Pascal Hitzler, Miriam Fernández, Krzysztof Janowicz, Amrapali Zaveri, Alasdair J. G. Gray, Vanessa López, Armin Haller, and Karl Hammar, editors, *The Semantic Web - 16th International Conference, ESWC 2019, Portorož, Slovenia, June 2-6, 2019, Proceedings*, volume 11503 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2019.

[AMH19b] Christian Aebeloe, Gabriela Montoya, and Katja Hose. Decentralized indexing over a network of RDF peers. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I*, volume 11778 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2019.

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422. ACM, 2007.

[AMMH09] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. Sw-store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.

[ANS18] Adnan Akhter, Axel-Cyrille Ngonga Ngomo, and Muhammad Saleem. An empirical evaluation of RDF graph partitioning techniques. In Catherine Faron-Zucker, Chiara Ghidini, Amedeo Napoli, and Yannick Toussaint, editors, *Knowledge Engineering and Knowledge Management - 21st International Conference, EKAW 2018, Nancy, France, November 12-16, 2018, Proceedings*, volume 11313 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2018.

[AÖDH13] Günes Aluç, M. Tamer Özsu, Khuzaima S. Daudjee, and Olaf Hartig. chameleon-db: a workload-aware robust rdf data management system. 2013.

[APU14] Carlos Buil Aranda, Axel Polleres, and Jürgen Umbrich. Strategies for executing federated queries in SPARQL1.1. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, volume 8797 of *Lecture Notes in Computer Science*, pages 390–405. Springer, 2014.

[ASY+21] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *VLDB*, abs/2102.13027, 2021.

[ATP20a] Amr Azzam, Ruben Taelman, and Axel Polleres. Towards cost-model-based query execution over hybrid linked data fragments interfaces. In Andreas Harth, Valentina Presutti, Raphaël Troncy, Maribel Acosta, Axel Polleres, Javier D. Fernández, Josiane Xavier Parreira, Olaf Hartig, Katja Hose, and Michael Cochez, editors, *The Semantic Web: ESWC 2020 Satellite Events - ESWC 2020 Satellite Events, Heraklion, Crete, Greece, May 31 - June 4, 2020, Revised Selected Papers*, volume 12124 of *Lecture Notes in Computer Science*, pages 9–12. Springer, 2020.

[ATP20b] Amr Azzam, Ruben Taelman, and Axel Polleres. Towards cost-model-based query execution over hybrid linked data fragments interfaces. In Andreas Harth, Valentina Presutti, Raphaël Troncy, Maribel Acosta, Axel Polleres, Javier D. Fernández, Josiane Xavier Parreira, Olaf Hartig, Katja Hose, and Michael Cochez, editors, *The Semantic Web: ESWC 2020 Satellite Events - ESWC 2020 Satellite Events, Heraklion, Crete, Greece, May 31 - June 4, 2020, Revised Selected Papers*, volume 12124 of *Lecture Notes in Computer Science*, pages 9–12. Springer, 2020.

[AV15] Maribel Acosta and Maria-Esther Vidal. Networks of linked data eddies: An adaptive web query processing engine for RDF data. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu

d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, volume 9366 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2015.

[Azz20] Amr Azzam. Enabling web-scale knowledge graphs querying. In Andreas Harth, Valentina Presutti, Raphaël Troncy, Maribel Acosta, Axel Polleres, Javier D. Fernández, Josiane Xavier Parreira, Olaf Hartig, Katja Hose, and Michael Cochez, editors, *The Semantic Web: ESWC 2020 Satellite Events - ESWC 2020 Satellite Events, Heraklion, Crete, Greece, May 31 - June 4, 2020, Revised Selected Papers*, volume 12124 of *Lecture Notes in Computer Science*, pages 229–239. Springer, 2020.

[BB08] Dave Beckett and Tim Berners-Lee. Turtle - terse RDF triple language, W3C team submission, 2008. See: http://www.w3.org/TeamSubmission/turtle/.

[BBD09] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. Progressive parametric query optimization. *IEEE Trans. Knowl. Data Eng.*, 21(4):582–594, 2009.

[BCK+08] Tim Berners-Lee, Dan Connolly, Lalana Kagal, Yosi Scharf, and Jim Hendler. N3logic: A logical framework for the world wide web. *Theory Pract. Log. Program.*, 8(3):249–269, 2008.

[BDPP18] Piero Andrea Bonatti, Stefan Decker, Axel Polleres, and Valentina Presutti. Knowledge graphs: New directions for knowledge representation on the semantic web (dagstuhl seminar 18371). *Dagstuhl Reports*, 8(9):29–111, 2018.

[BE12] C Glenn Begley and Lee M Ellis. Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, 2012.

[BEP+08] Kurt D. Bollacker, Colin Evans, Praveen K. Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1247–1250. ACM, 2008.

[Ber14] David Bernstein. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Comput.*, 1(3):81–84, 2014.

[BF00a] Tim Berners-Lee and Mark Fischetti. *Weaving the web - the original design and ultimate destiny of the World Wide Web by its inventor*. HarperBusiness, 2000.

[BF00b] Tim Berners-Lee and Mark Fischetti. *Weaving the web - the original design and ultimate destiny of the World Wide Web by its inventor*. HarperBusiness, 2000.

[BG04] Wolf-Tilo Balke and Ulrich Güntzer. Multi-objective query processing for database systems. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 936–947. Morgan Kaufmann, 2004.

[BHB09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[BL98] Tim Berners-Lee. Semantic Web Road map. Website, 1998. http://www.w3.org/DesignIssues/Semantic.html.

[BL06] Tim Berners-Lee. Linked data - design issues. *W3C*, 2006.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, 284:34–43, 2001.

[BMT17] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *Proc. VLDB Endow.*, 11(2):149–161, 2017.

[Boe15] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Oper. Syst. Rev.*, 49(1):71–79, 2015.

[BS09] Christian Bizer and Andreas Schultz. The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[CCZZ14] Xi Chen, Huajun Chen, Ningyu Zhang, and Songyang Zhang. Sparkrdf: Elastic discreted RDF graph processing engine with distributed memory. In Matthew Horridge, Marco Rospocher, and Jacco van Ossenbruggen, editors, *Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014*, volume 1272 of *CEUR Workshop Proceedings*, pages 261–264. CEUR-WS.org, 2014.

[CF04] Min Cai and Martin R. Frank. Rdfpeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 650–657. ACM, 2004.

[CFL18] Matteo Cossu, Michael Färber, and Georg Lausen. Prost: Distributed execution of SPARQL queries using mixed partitioning strategies. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 469–472. OpenProceedings.org, 2018.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, page 34–43, New York, NY, USA, 1998. Association for Computing Machinery.

[CHI15] Tom Crick, Benjamin A. Hall, and Samin S. Ishtiaq. Reproducibility in research: Systems, infrastructure, culture. *Journal of Open Research Software*, 2015.

[CK92] Jon F Claerbout and Martin Karrenbach. Electronic documents give reproducible research a new meaning. In *SEG technical program expanded abstracts 1992*, pages 601–604. Society of Exploration Geophysicists, 1992.

[CL18] Xiaowei Chen and John C. S. Lui. Mining graphlet counts in online social networks. *ACM Trans. Knowl. Discov. Data*, 12(4):41:1–41:38, 2018.

[CW06] Surajit Chaudhuri and Gerhard Weikum. Foundations of automated database tuning. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 104. IEEE Computer Society, 2006.

[dBFT05] Jos de Bruijn, Enrico Franconi, and Sergio Tessaris. Logical reconstruction of RDF and ontology languages. In François Fages and Sylvain Soliman, editors, *Principles and Practice of Semantic Web Reasoning, Third International Workshop, PPSWR 2005, Dagstuhl Castle, Germany, September 11-16, 2005, Proceedings*, volume 3703 of *Lecture Notes in Computer Science*, pages 65–71. Springer, 2005.

[DCC+14] Michel Dumontier, Alison Callahan, Jose Cruz-Toledo, Peter Ansell, Vincent Emonet, François Belleau, and Arnaud Droit. Bio2rdf release 3: A larger, more connected network of linked data for the life sciences. In Matthew Horridge, Marco Rospocher, and Jacco van Ossenbruggen, editors, *Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC*

2014, Riva del Garda, Italy, October 21, 2014, volume 1272 of *CEUR Workshop Proceedings*, pages 401–404. CEUR-WS.org, 2014.

[DÇPU11] Jennie Duggan, Ugur Çetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 337–348. ACM, 2011.

[DGK+15] Benjamin Djahandideh, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. Cliquesquare in action: Flat plans for massively parallel RDF queries. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1432–1435. IEEE Computer Society, 2015.

[DKSU11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 145–156. ACM, 2011.

[DMvH+00] Stefan Decker, Sergey Melnik, Frank van Harmelen, Dieter Fensel, Michel C. A. Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of XML and RDF. *IEEE Internet Comput.*, 4(5):63–74, 2000.

[DMY+20] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. ALEX: an updatable adaptive learned index. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 969–984. ACM, 2020.

[DWNY12] Jin-Hang Du, Haofen Wang, Yuan Ni, and Yong Yu. Hadooprdf: A scalable semantic data analytical engine. In De-Shuang Huang, Jianhua Ma, Kang-Hyun Jo, and M. Michael Gromiha, editors, *Intelligent Computing Theories and Applications - 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings*, volume 7390 of *Lecture Notes in Computer Science*, pages 633–641. Springer, 2012.

[EM09] Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge - Networked Media - Integrating Knowledge Management, New Media Technologies and Semantic Systems*, volume 221 of *Studies in Computational Intelligence*, pages 7–24. 2009.

[ESW15] Patrick Ernst, Amy Siu, and Gerhard Weikum. Knowlife: a versatile approach for constructing a large knowledge graph for biomedical sciences. *BMC Bioinform.*, 16:157:1–157:13, 2015.

[Fel98] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[FJK96] Michael J. Franklin, Björn Þór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 149–160. ACM Press, 1996.

[FLM98] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Rec.*, 27(3):59–74, 1998.

[FMG+13] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutierrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (HDT). *J. Web Semant.*, 19:22–41, 2013.

[FMPdlFRG18] Javier D Fernández, Miguel A Martínez-Prieto, Pablo de la Fuente Redondo, and Claudio Gutiérrez. Characterising rdf data sets. *J. Inf. Sci.*, 44(2):203–229, apr 2018.

[FMPR18] Javier D. Fernández, Miguel A. Martínez-Prieto, Axel Polleres, and Julian Reindorf. HDTQ: managing RDF datasets in compressed space. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, volume 10843 of *Lecture Notes in Computer Science*, pages 191–208. Springer, 2018.

[FSMM15] Pauline Folz, Hala Skaf-Molli, and Pascal Molli. Cyclades: A decentralized cache for linked data fragments. In *Extended Semantic Web Conference*, 2015.

[FWCT13] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. SPARQL 1.1 protocol. *Recommendation, W3C, March*, 2013.

[Gan98] Sumit Ganguly. Design and analysis of parametric query optimization algorithms. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 228–238. Morgan Kaufmann, 1998.

[GFI16] Steven N. Goodman, Daniele Fanelli, and John P. A. Ioannidis. What does research reproducibility mean? *Science Translational Medicine*, 8:341ps12 – 341ps12, 2016.

[GGvHS10] Christophe Guéret, Paul Groth, Frank van Harmelen, and Stefan Schlobach. Finding the achilles heel of the web of data: Using network analysis for link-recommendation. In *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, volume 6496 of *Lecture Notes in Computer Science*, pages 289–304. Springer, 2010.

[GHMP11] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. Foundations of semantic web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.

[GHS14] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: a distributed engine for efficient RDF processing. In Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel, editors, *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 267–268. ACM, 2014.

[GJGL16] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. SPARQLGX in action: Efficient distributed evaluation of SPARQL with apache spark. In Takahiro Kawamura and Heiko Paulheim, editors, *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016*, volume 1690 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[GM93] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993.

[GN14] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy, editors, *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014,*

Athens, Greece, March 24-28, 2014, pages 439–450. OpenProceedings.org, 2014.

[Gra95] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[GSMT14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 289–300. ACM, 2014.

[HA16] Olaf Hartig and Carlos Buil Aranda. Bindings-restricted triple pattern fragments. In Christophe Debruyne, Hervé Panetto, Robert Meersman, Tharam S. Dillon, Eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*, volume 10033 of *Lecture Notes in Computer Science*, pages 762–779, 2016.

[HA20] L. Heling and M. Acosta. Cost- and robustness-based query optimization for linked data fragments. In *ISWC 2020*, pages 238–257, 2020.

[HA22] Lars Heling and Maribel Acosta. Federated SPARQL query processing over heterogeneous linked data fragments. In Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini, editors, *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, pages 1047–1057. ACM, 2022.

[HAMS18] Lars Heling, Maribel Acosta, Maria Maleshkova, and York Sure-Vetter. Querying large knowledge graphs over triple pattern fragments: An empirical study. In Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, volume 11137 of *Lecture Notes in Computer Science*, pages 86–102. Springer, 2018.

[Har11a] A. Harth. Cumulusrdf: Linked data management on nested key-value stores. 2011.

[HAR11b] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endow.*, 4(11):1123–1134, 2011.

[har12] Must try harder. *Nature*, 483(7391):509–509, Mar 2012.

[Har13] Olaf Hartig. SQUIN: a traversal based query execution system for the web of linked data. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1081–1084. ACM, 2013.

[HB11] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[HFKP20] Armin Haller, Javier D. Fernández, Maulik R. Kamdar, and Axel Polleres. What are links in linked open data? a characterization and evaluation of links between knowledge graphs on the web. *J. Data and Information Quality*, 12(2), may 2020.

[HKH+14] Ali Hasnain, Maulik R. Kamdar, Panagiotis Hasapis, Dimitris Zeginis, Claude N. Warren Jr., Helena F. Deus, Dimitrios Ntalaperas, Konstantinos A. Tarabanis, Muntazir Mehdi, and Stefan Decker. Linked biomedical dataspace: Lessons learned integrating data for drug discovery. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2014.

[HKWY97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 276–285. Morgan Kaufmann, 1997.

[HLY93] Kien A. Hua, Yu-lung Lo, and Honesty C. Young. Considering data skew factor in multi-way join query optimization for parallel execution. *VLDB J.*, 2(3):303–330, 1993.

[HMF15] Antonio Hernández-Illera, Miguel A. Martínez-Prieto, and Javier D. Fernández. Serializing RDF in compressed space. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pages 363–372. IEEE, 2015.

[Hog17] Aidan Hogan. Canonical forms for isomorphic and equivalent rdf graphs: Algorithms for leaning and labelling blank nodes. *ACM Trans. Web*, 11(4), jul 2017.

[HPS14] P. Hayes and P. Patel-Schneider. RDF 1.1 semantics, 2014. https://www.w3.org/TR/rdf11-mt/.

[HRW+20] S. M. Shamimul Hasan, Donna R. Rivera, Xiao-Cheng Wu, Eric B. Durbin, James Blair Christian, and Georgia D. Tourassi. Knowledge graph-enabled cancer data analytics. *IEEE J. Biomed. Health Informatics*, 24(7):1952–1967, 2020.

[HS02] Arvind Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 167–178. Morgan Kaufmann, 2002.

[HS13] Katja Hose and Ralf Schenkel. WARP: workload-aware replication and partitioning for RDF. In Chee Yong Chan, Jiaheng Lu, Kjetil Nørvåg, and Egemen Tanin, editors, *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1–6. IEEE Computer Society, 2013.

[HSBW13] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[HUHD07] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2007.

[HVMdW15] Joachim Van Herwegen, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. Query execution optimization for clients of triple pattern fragments. In Fabien Gandon, Marta Sabou, Harald Sack, Claudia d'Amato, Philippe Cudré-Mauroux, and Antoine Zimmermann, editors, *The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings*, volume 9088 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2015.

[HVV+15] Joachim Van Herwegen, Laurens De Vocht, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. Substring filtering for low-cost linked data interfaces. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth,

Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, volume 9366 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2015.

[IHPZ16] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Optimizing aggregate SPARQL queries using materialized RDF views. In Paul Groth, Elena Simperl, Alasdair J. G. Gray, Marta Sabou, Markus Krötzsch, Freddy Lécué, Fabian Flöck, and Yolanda Gil, editors, *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I*, volume 9981 of *Lecture Notes in Computer Science*, pages 341–359, 2016.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, sep 1984.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, mar 1996.

[JST17] Daniel Janke, Steffen Staab, and Matthias Thimm. Koral: A glass box profiling system for individual components of distributed RDF stores. In Ricardo Usbeck, Axel-Cyrille Ngonga Ngomo, Jin-Dong Kim, Key-Sun Choi, Philipp Cimiano, Irini Fundulaki, and Anastasia Krithara, editors, *Joint Proceedings of BLINK2017: 2nd International Workshop on Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 21st - to - 22nd, 2017*, volume 1932 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[Kam19] Maulik R Kamdar. *A web-based integration framework over heterogeneous biomedical data and knowledge sources*. Stanford University, 2019.

[KBC+18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[KCC+21] Sunghwan Kim, Jie Chen, Tiejun Cheng, Asta Gindulyte, Jia He, Siqian He, Qingliang Li, Benjamin A. Shoemaker, Paul A. Thiessen, Bo Yu, Leonid Zaslavsky, Jian Zhang, and Evan Bolton. Pubchem in 2021:

new data content and improved web interfaces. *Nucleic Acids Res.*, 49(Database-Issue):D1388–D1395, 2021.

[KDZ+17] Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6348–6358, 2017.

[KFP+19] Maulik R. Kamdar, Javier D. Fernández, Axel Polleres, Tania Tudorache, and Mark A. Musen. Enabling web-scale data integration in biomedicine through linked open data. *npj Digital Medicine*, 2(1):90, September 2019.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[KKK+10] Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Iris Miliaraki, Matoula Magiridou, and Antonios Papadakis-Pesaresi. Atlas: Storing, updating and querying RDF(S) data on top of dhts. *J. Web Semant.*, 8(4):271–277, 2010.

[KM15] Zoi Kaoudi and Ioana Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015.

[KSR+07] Marcel Karnstedt, Kai-Uwe Sattler, Martin Richtarsky, Jessica Müller, Manfred Hauswirth, Roman Schmidt, and Renault John. Unistore: Querying a dht-based universal storage. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1503–1504. IEEE Computer Society, 2007.

[KSR+09] Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizer, and Robert Lee. Media meets semantic web - how the BBC uses dbpedia and linked data to make connections. In Lora Aroyo, Paolo Traverso, Fabio Ciravegna, Philipp Cimiano, Tom Heath, Eero Hyvönen, Riichiro Mizoguchi, Eyal Oren, Marta Sabou, and Elena Paslaru Bontas Simperl, editors, *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings*, volume 5554 of *Lecture Notes in Computer Science*, pages 723–737. Springer, 2009.

[KSTI11] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis E. Ioannidis. Schedule optimization for data processing flows on the cloud. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 289–300. ACM, 2011.

[KYG+18] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[Ley02] Michael Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. In Alberto H. F. Laender and Arlindo L. Oliveira, editors, *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*, volume 2476 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2002.

[LG12] Markus Lanthaler and Christian Gütl. On using JSON-LD to create evolvable restful services. In Rosa Alarcón, Cesare Pautasso, and Erik Wilde, editors, *Third International Workshop on RESTful Design, WS-REST '12, Lyon, France, April 16, 2012*, pages 25–32. ACM, 2012.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[LIJ+15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[LL13] Kisung Lee and Ling Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *Proc. VLDB Endow.*, 6(14):1894–1905, 2013.

[LLWL08] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Irlbot: scaling to 6 billion pages and beyond. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 427–436. ACM, 2008.

[LSB+17] Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Ivan Ermilov, Simon Bin, Nilesh Chakraborty, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, and Hajira Jabeen. Distributed

semantic analytics using the SANSA stack. In Claudia d'Amato, Miriam Fernández, Valentina A. M. Tamma, Freddy Lécué, Philippe Cudré-Mauroux, Juan F. Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II*, volume 10588 of *Lecture Notes in Computer Science*, pages 147–155. Springer, 2017.

[LWYZ19] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, 2019.

[LZLG19] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, 2019.

[MAA18] Amgad Madkour, Ahmed M. Aly, and Walid G. Aref. WORQ: workload-driven RDF query processing. In Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I*, volume 11136 of *Lecture Notes in Computer Science*, pages 583–599. Springer, 2018.

[MAH+18] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 631–645. ACM, 2018.

[MBC+16] Miguel A. Martínez-Prieto, Nieves R. Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Inf. Syst.*, 56:73–108, 2016.

[MBK02] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 191–202. Morgan Kaufmann, 2002.

[McB01] Brian McBride. Jena: Implementing the RDF model and syntax specification. In Stefan Decker, Dieter A. Fensel, Amit P. Sheth, and Steffen Staab, editors, *Proceedings of the Second International Workshop on the Semantic Web - SemWeb'2001, Hongkong, China, May 1, 2001*, volume 40 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[MGF12] Miguel A. Martínez-Prieto, Mario Arias Gallego, and Javier D. Fernández. Exchange and consumption of huge RDF data. In Elena Simperl, Philipp Cimiano, Axel Polleres, Óscar Corcho, and Valentina Presutti, editors, *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, volume 7295 of *Lecture Notes in Computer Science*, pages 437–452. Springer, 2012.

[MKH19] Gabriela Montoya, Ilkcan Keles, and Katja Hose. Analysis of the effect of query shapes on performance over LDF interfaces. In Muhammad Saleem, Aidan Hogan, Ricardo Usbeck, Axel-Cyrille Ngonga Ngomo, and Ruben Verborgh, editors, *Proceedings of the QuWeDa 2019: 3rd Workshop on Querying and Benchmarking the Web of Data co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26-30, 2019*, volume 2496 of *CEUR Workshop Proceedings*, pages 51–66. CEUR-WS.org, 2019.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 149–159. Morgan Kaufmann, 1986.

[MLAN11] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia SPARQL benchmark - performance assessment with real queries on real data. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.

[MNM+19] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[MP18] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In Rajesh Bordawekar and Oded Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.

[MP19a] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *9th Biennial Conference on Innova-*

tive Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings. www.cidrdb.org, 2019.

[MP19b] Ryan C. Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, 2019.

[MPMA17] Marios Meimaris, George Papastefanatos, Nikos Mamoulis, and Ioannis Anagnostopoulos. Extended characteristic sets: Graph indexing for SPARQL query optimization. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 497–508. IEEE Computer Society, 2017.

[MPS+17] Antonio Messina, Haikal Pribadi, Jo Stichbury, Michelangelo Bucci, Szymon Klarman, and Alfonso Urso. Biograkn: A knowledge graph-based semantic database for biomedical sciences. In Leonard Barolli and Olivier Terzo, editors, *Complex, Intelligent, and Software Intensive Systems - Proceedings of the 11th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS-2017), Torino, Italy, July 10-12, 2017*, volume 611 of *Advances in Intelligent Systems and Computing*, pages 299–309. Springer, 2017.

[MSM19] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. Sage: Web preemption for public SPARQL query services. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 1268–1278. ACM, 2019.

[MVC+12] Gabriela Montoya, Maria-Esther Vidal, Óscar Corcho, Edna Ruckhaus, and Carlos Buil Aranda. Benchmarking federated SPARQL query engines: Are existing testbeds enough? In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, volume 7650 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2012.

[NCG18] Yaroslav Nechaev, Francesco Corcoglioniti, and Claudio Giuliano. Sociallink: exploiting graph embeddings to link dbpedia entities to twitter profiles. *Prog. Artif. Intell.*, 7(4):251–272, 2018.

[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors,

Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, pages 984–994. IEEE Computer Society, 2011.

[OBGK18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In Sebastian Schelter, Stephan Seufert, and Arun Kumar, editors, *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 4:1–4:4. ACM, 2018.

[PAA+17] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[PKB+20] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1099–1114. ACM, 2020.

[PKF+20] Axel Polleres, Maulik R. Kamdar, Javier D. Fernández, Tania Tudorache, and Mark A. Musen. A more decentralized vision for linked data. *Semantic Web*, 11(1):101–113, 2020.

[RHSG14] Laurens Rietveld, Rinke Hoekstra, Stefan Schlobach, and Christophe Guéret. Structural properties as proxy for semantic relevance in RDF graph sampling. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, volume 8797 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2014.

[RRR22] Wullianallur Raghupathi, Viju Raghupathi, and Jie Ren. Reproducibility in computing research: An empirical study. *IEEE Access*, 10:29207–29223, 2022.

[RS10] Kurt Rohloff and Richard E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the SHARD triple-store. In Eli Tilevich and Patrick Eugster, editors, *SPLASH Workshop on Programming Support Innovations for Emerging Distributed Applications (PSI EtA - Ψ 2010), October 17, 2010, Reno/Tahoe, Nevada, USA*, page 4. ACM, 2010.

[SAH+15] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: the linked SPARQL queries dataset. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, volume 9367 of *Lecture Notes in Computer Science*, pages 261–269. Springer, 2015.

[SHA+12] Manuel Salvadores, Matthew Horridge, Paul R. Alexander, Ray W. Fergerson, Mark A. Musen, and Natalya Fridman Noy. Using SPARQL to query bioportal ontologies and metadata. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, volume 7650 of *Lecture Notes in Computer Science*, pages 180–195. Springer, 2012.

[SHLP09] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Sp^2bench: A SPARQL performance benchmark. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 222–233. IEEE Computer Society, 2009.

[SHN18] Muhammad Saleem, Ali Hasnain, and Axel-Cyrille Ngonga Ngomo. Largerdfbench: A billion triples benchmark for SPARQL endpoint federation. *J. Web Semant.*, 48:85–125, 2018.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 697–706. ACM, 2007.

[SL19] Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.

[SMK18] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 1043–1052. ACM, 2018.

[SML10] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In Luc Segoufin, editor, *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, ACM International Conference Proceeding Series, pages 4–33. ACM, 2010.

[SMN15] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. FEASIBLE: A feature-based SPARQL benchmark generation framework. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, volume 9366 of *Lecture Notes in Computer Science*, pages 52–69. Springer, 2015.

[SPL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. Pigsparql: mapping SPARQL to pig latin. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, Athens, Greece, June 12, 2011*, page 4. ACM, 2011.

[SPNL14] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. Sempala: Interactive SPARQL query processing on hadoop. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 164–179. Springer, 2014.

[SPSL16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on spark. *Proc. VLDB Endow.*, 9(10):804–815, 2016.

[SR14] G. Schreiber and Y. Raimond. *RDF 1.1 Primer*. W3C Working Group Note, 2014. https://www.w3.org/TR/rdf11-primer/.

[SS04] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.

[SVH+15] Miel Vander Sande, Ruben Verborgh, Joachim Van Herwegen, Erik Mannens, and Rik Van de Walle. Opportunistic linked data querying through approximate membership metadata. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, volume 9366 of *Lecture Notes in Computer Science*, pages 92–110. Springer, 2015.

[THSV18] Ruben Taelman, Joachim Van Herwegen, Miel Vander Sande, and Ruben Verborgh. Comunica: A modular SPARQL query engine for the web. In Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, volume 11137 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2018.

[TK14] Immanuel Trummer and Christoph Koch. Approximation schemes for many-objective query optimization. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1299–1310. ACM, 2014.

[TK17] Immanuel Trummer and Christoph Koch. Multi-objective parametric query optimization. *Commun. ACM*, 60(10):81–89, 2017.

[TSVM17] Ruben Taelman, Miel Vander Sande, Ruben Verborgh, and Erik Mannens. Versioned triple pattern fragments: A low-cost linked data interface feature for web archives. In Jeremy Debattista, Jürgen Umbrich, Javier D. Fernández, Anisa Rula, Amrapali Zaveri, Anastasia Dimou, and Wouter Beek, editors, *Joint proceedings of the 3rd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW 2017) and the 4th Workshop on Linked Data Quality (LDQ 2017) co-located with 14th European Semantic Web Conference (ESWC 2017), Portorož, Slovenia, May 28th-29th, 2017*, volume 1824 of *CEUR Workshop Proceedings*, pages 1–11. CEUR-WS.org, 2017.

[TVCM16] Ruben Taelman, Ruben Verborgh, Pieter Colpaert, and Erik Mannens. Continuous client-side query evaluation over dynamic linked data. In Harald Sack, Giuseppe Rizzo, Nadine Steinmetz, Dunja Mladenic, Sören Auer, and Christoph Lange, editors, *The Semantic Web - ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May 29 - June 2, 2016, Revised Selected Papers*, volume 9989 of *Lecture Notes in Computer Science*, pages 273–289, 2016.

[TWW+21] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Trans. Database Syst.*, 46(3):9:1–9:45, 2021.

[VK14] Denny Vrandecic and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[VMZC15] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. Join size estimation subject to filter conditions. *Proc. VLDB Endow.*, 8(12):1530–1541, 2015.

[VSH+16] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. Triple pattern fragments: A low-cost knowledge graph interface for the web. *J. Web Semant.*, 37-38:184–206, 2016.

[VUM+17] Pierre-Yves Vandenbussche, Jürgen Umbrich, Luca Matteis, Aidan Hogan, and Carlos Buil Aranda. SPARQLES: monitoring public SPARQL endpoints. *Semantic Web*, 8(6):1049–1065, 2017.

[WCHN13] Wentao Wu, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proc. VLDB Endow.*, 6(10):925–936, 2013.

[WKG+08] David S. Wishart, Craig Knox, Anchi Guo, Dean Cheng, Savita Shrivastava, Dan Tzur, Bijaya Gautam, and Murtaza Hassanali. Drugbank: a knowledgebase for drugs, drug actions and drug targets. *Nucleic Acids Res.*, 36(Database-Issue):901–906, 2008.

[WMPH19] Randall T. Whitman, Bryan G. Marsh, Michael B. Park, and Erik G. Hoel. Distributed spatial and spatio-temporal join on apache spark. *ACM Trans. Spatial Algorithms Syst.*, 5(1):6:1–6:28, 2019.

[WZC+16] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 45(2):17–22, 2016.

[YLCL21] Zhengtong Yan, Jiaheng Lu, Naresh Chainani, and Chunbin Lin. Workload-aware performance tuning for autonomous dbmss. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 2365–2368. IEEE, 2021.

[YYG19] James Jian Qiao Yu, Wen Yu, and Jiatao Gu. Online vehicle routing with neural combinatorial optimization and deep reinforcement learning. *IEEE Trans. Intell. Transp. Syst.*, 20(10):3806–3817, 2019.

[ZCTW13] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 565–576. IEEE Computer Society, 2013.

[ZLZ+19] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 415–432. ACM, 2019.

[ZMBR20] Shuo Zhang, Edgar Meij, Krisztian Balog, and Ridho Reinanda. Novel entity discovery from web tables. In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 1298–1308. ACM / IW3C2, 2020.

[ZMG+20] Ishaq Zouaghi, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, and Taoufik Aguili. Query optimization for large scale clustered RDF data. In Il-Yeol Song, Katja Hose, and Oscar Romero, editors, *Proceedings of the 22nd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with EDBT/ICDT 2020 Joint Conference, DOLAP@EDBT/ICDT 2020, Copenhagen, Denmark, March 30, 2020*, volume 2572 of *CEUR Workshop Proceedings*, pages 56–65. CEUR-WS.org, 2020.

[ZSLF20] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow.*, 13(9):1416–1428, 2020.