Decentralized Knowledge Graphs on the Web

Aebeloe, Christian

DOI (link to publication from Publisher):
10.54337/aau495046844

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

DECENTRALIZED KNOWLEDGE GRAPHS ON THE WEB

BY
CHRISTIAN AEBELOE

DISSERTATION SUBMITTED 2022

AALBORG UNIVERSITY
DENMARK

Decentralized Knowledge Graphs on the Web

Ph.D. Dissertation

Christian Aebeloe

Dissertation submitted July, 2022

Dissertation submitted: July, 2022

PhD supervisor: Professor Katja Hose
Aalborg University

PhD Co-Supervisor: Associate Professor Gabriela Montoya
Aalborg University

PhD committee: Associate Professor Michele Albano (chairman)
Aalborg University, Denmark

Associate Professor Aidan Hogan
University of Chile, Chile

Professor Ruben Verborgh
Ghent University, Belgium

PhD Series: Technical Faculty of IT and Design, Aalborg University

Department: Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-863-2

Published by:
Aalborg University Press
Kroghstræde 3
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Stibo Complete, 2022

Abstract

The increasing popularity of the Web of Data has over the past years caused a significant increase in the number of open knowledge graphs published on the Web. However, the structure of the Semantic Web today relies totally on the data providers to maintain Web services that provide efficient and scalable access to the knowledge graphs, as well as to keep the data up-to-date. Without any monetary incentives to do so, publicly available knowledge graphs often become unavailable and outdated, making it difficult to trust the data available in the knowledge graphs. As a remedy, this thesis investigates and addresses the availability, scalability, and updatability issues with the aim of making knowledge graphs on the Web available, scalable, and updatable.

First, the thesis explores load-balancing in client-server architectures in order to increase the availability and scalability of the knowledge graphs under heavy query processing load by introducing a system that we call WiseKG. To optimize queries in such a setup, WiseKG decomposes the query into star-shaped subqueries and determines, using a cost model that considers factors such as the current load on the server and the data transfer overhead, whether each subquery can be processed more efficiently by the client or the server. A comprehensive experimental evaluation shows that WiseKG significantly improves query processing performance for high demanding workloads compared to state-of-the-art systems while being able to answer more queries without timing out.

Second, the thesis addresses the availability and scalability issues from a different point of view by proposing a decentralized Peer-to-Peer (P2P) architecture for sharing and querying knowledge graphs, called Piqnic. In Piqnic, nodes act as both clients and servers and thus maintain a local datastore (a set of knowledge graph fragments) and a local view over the network (a set of neighboring nodes). Piqnic replicates the fragments over multiple nodes within the network, ensuring the availability of the data even if the uploading node fails. An experimental evaluation shows that by doing so, Piqnic maintains high availability even when a large part of the nodes fails.

Third, the thesis introduces two novel indexing schemes that allow nodes

to determine which nodes hold relevant data to which subquery: (1) baseline *locational* indexes that map each predicate in the query to the nodes that hold relevant data to each predicate, and (2) *Prefix-Partitioned Bloom Filter* indexes that represent the set of subjects and objects in a fragment as a partitioned bitvector that allows the nodes to ascertain the joinability of two fragments. An experimental evaluation supports the hypothesis that such indexes significantly improve query processing performance while decreasing the network usage compared to Piqnic.

Fourth, the thesis addresses the updatability issue by introducing ColChain, a system that allows the users to collaborate on keeping the data up to date. ColChain divides the P2P network into *communities* of nodes and relies on community-wide consensus to allow nodes to propose and apply consensual updates to the fragments. Furthermore, ColChain represents the entire history of updates to a fragment as a *chain* of updates, allowing nodes to trace-back faulty updates to their origin, as well as to dynamically roll-back fragments to an earlier version and process queries over them. A comprehensive experimental evaluation shows that ColChain provides efficient community-wide consensual updates to the fragments without incurring a significant cost on query performance.

Fifth, the thesis demonstrates ColChain and introduces a fully functioning ColChain client with a graphical UI. The demonstration highlights how users can navigate the fragments stored by a particular ColChain node. Furthermore, the demonstration shows how users can participate in keeping the fragments up to date, as well as how queries can be processed over a previous version of the fragments.

Last, the thesis introduces the Lothbrok approach to optimizing SPARQL queries in the decentralized setup. In particular, Lothbrok fragments data based on characteristic sets (predicate families) and uses star-shaped query decomposition similar to WiseKG. To accommodate the fragmentation technique, Lothbrok further introduces a novel indexing scheme, called *Semantically Partitioned Bloom Filter* indexes, that associates the objects in a fragment with the predicates they occur in triples with. Lothbrok nodes use these indexes to build a query execution plan in consideration of cardinality estimations, the compatibility of fragments for the given query, and the locality of the data. Lothbrok is further able to delegate subqueries to other nodes in the network such that the network overhead is minimized. A comprehensive experimental study shows a performance increase of up to two orders of magnitude when comparing Lothbrok to Piqnic and ColChain while the network usage is lowered as well.

Resumé

Det Semantiske Webs stigende popularitet har over de seneste år forårsaget en væsentlig stigning i antallet af frit tilgængelige vidensgrafer på nettet. Strukturen af det Semantiske Web afhænger imidlertid i dag af, at dataudbyderne vedligeholder webtjenester, der giver effektiv og skalerbar adgang til vidensgraferne, samt holder dataen opdateret. Manglen på monetære tilkyndelser til at vedligeholde vidensgrafer er i høj grad skyld i at frit tilgængelige vidensgrafer ofte er utilgængelige og forældede, hvilket gør det svært at stole på den tilgængelige data. Denne afhandling undersøger løsninger til hvert af disse problemer individuelt med formålet om at gøre åbne vidensgrafer mere tilgængelige, skalerbare og opdaterbare.

For det første udforsker denne afhandling belastningsbalancering i klient-server arkitekturer for at øge tilgængeligheden og skalerbarheden af vidensgraferne, når mange forespørgsler bliver behandlet samtidigt, og introducerer et system kaldet WiseKG. For at effektivisere behandlingen af forespørgsler i sådan et setup, nedbryder WiseKG forespørgslen til stjerneformede underforespørgsler og fastlægger, baseret på faktorer såsom den nuværende belastning på serveren samt overhead af dataoverførsel, om hver forespørgsel kan behandles mest effektivt af klienten eller serveren. En omfattende undersøgelse viser, at WiseKG forbedrer ydeevnen for behandling af forespørgsler væsentligt for højt krævende arbejdsbyrder sammenlignet med state-of-the-art systemer, samt kan besvare flere forespørgsler uden at time ud.

For det andet undersøger afhandlingen tilgængeligheds- og skalerbahedsproblemerne fra et andet perspektiv ved at foreslå en decentraliseret Peer-to-Peer (P2P) arkitektur til deling og forespørgselsbehandling af vidensgrafer, kaldet Piqnic. Noder i Piqnic agerer både som klienter og servere, og vedligeholder derfor et lokalt datalager (en mængde af vidensgraffragmenter), samt en lokal oversigt over netværket (en mængde af nabonoder). Piqnic replikerer fragmenterne på flere noder i netværket for at sikre tilgængeligheden af dataen selv i tilfælde af, at den uploadende node fejler. En eksperimentel undersøgelse viser, at Piqnic fastholder høj tilgængelighed, selv når en stor andel af noderne fejler.

For det tredje introducerer afhandlingen en ny form for decentraliseret in-

deks, der tillader noder i et P2P-netværk at fastslå præcis hvilke noder, der indeholder relevant data til hvilke underforspørgsler: (1) *locational* indekser, der matcher hvert prædikat i forespørgslen med de noder, der indeholder relevant data for dem, og (2) *Prefix-Partitioned Bloom Filter*-indekser der repræsenterer mængden af subjekter og objekter i et fragment som en opdelt bitvektor, der lader noderne fastslå om to fragmenter producerer join-resultater for en given forespørgsel. En eksperimentel undersøgelse understøtter hypotesen om, at sådanne indekser forbedrer ydeevnen for behandling af forespørgsler væsentligt ved at formindske belastningen på netværket sammenlignet med Piqnic.

For det fjerde adresserer afhandlingen opdaterbarhedsproblemet ved at introducere ColChain, et system der tillader brugerne at samarbejde om at holde dataen opdateret. ColChain deler netværket op i mindre *fællesskaber* af noder og bruger konsensus over fællesskaberne til at lade noder i netværket foreslå og anvende konsensuelle opdateringer til vidensgraferne. Ydermere repræsenterer ColChain hele opdateringshistorikken af et fragment som en *kæde* af opdateringer, hvilket lader brugerne spore fejlslagne opdateringer til deres oprindelse, samt dynamisk at behandle forespørgsler over tidligere tilgængelige versioner af vidensgraferne. En omfattende undersøgelse viser, at ColChain tillader effektive konsensuelle opdateringer af fragmenterne, foruden at pådrage sig en unødvendig omkostning på ydeevnen for behandling af forespørgsler.

For det femte demonstrerer afhandlingen ColChain ved at introducere en fuldt fungerende ColChain-klient med en grafisk brugerflade. Demonstrationen fremhæver, hvordan brugere kan navigere i vidensgraferne lagret af en bestemt ColChain-node. Desuden viser demonstrationen, hvordan brugere kan deltage i at holde vidensgraferne opdateret, samt hvordan forespørgsler kan behandles over tidligere versioner af vidensgraferne.

Til sidst introducerer afhandlingen Lothbrok, en metode til at optimere SPARQL-forespørgsler i den decentraliserede kontekst. Specifikt fragmenterer Lothbrok vidensgraferne baseret på *characteristic sets* (prædikatfamilier) og bruger stjerneformet nedbrydelse af forespørgslerne lignende WiseKG. Lothbrok introducerer ydermere en ny måde at indeksere dataen på ved at introducere *Semantically Partitioned Bloom Filter*-indekser, som associerer objekterne i et fragment med de prædikater, de optræder i tripler med. Lothbrok-noder bruger disse indekser til at bygge en plan for behandling af forespørgslerne, der tager højde for estimerede kardinaliteter, kompatibiliteten af fragmenter for en given forespørgsel, samt lokaliteten af dataen. Lothbrok-noder kan desuden delegere underforespørgsler til andre noder, såfremt det minimerer belastningen på netværket. En omfattende eksperimentel undersøgelse viser, at Lothbrok forbedrer effektiviteten for behandling af forespørgsler med op til to størrelsesordener sammenlignet med Piqnic og ColChain, samt at netværksbrugen også er formindsket.

Acknowledgments

First and foremost, I would like to thank my Ph.D. supervisors, Katja Hose and Gabriela Montoya, for providing helpful feedback and comments on drafts, for their patience despite my many questions and mistakes, and for their support and understanding throughout even the most difficult parts of my journey. I admire their dedication and attention to detail, without which this thesis and my publications would not be possible.

Secondly, I would like to thank my amazing girlfriend, Sabine Tovgaard, for her unwavering support throughout this journey. I know that it has not always been easy, but she has always stood besides me and provided me with all the support, comfort, and motivation I needed to continue. For that, I admire her greatly; this thesis would not have been possible without her.

My colleagues from the Data, Knowledge and Web Engineering (DKW) group at Cassiopeia, Aalborg University also deserve a huge thanks for creating a pleasant work environment throughout my employment here. They have managed to create a great work environment with room for both fun and cozy work environment, and challenging research interactions. Through board game events on Wednesdays and interesting conversations in lunch breaks, they have made my time here a true pleasure. I feel honored to have worked alongside such kind, dedicated, and intelligent people. Furthermore, I wish to thank the administrative staff at Aalborg University for their patience with me and assistance despite my many, and in many cases redundant, questions.

I also wish to extend my gratitude to the co-authors of all the papers I have been a part of during my time here at the University, not just the ones included in this thesis. Besides Katja and Gabriela, I thus thank Ilkcan Keles, Amr Azzam, Axel Polleres, and Vinay Setty, for their valuable input and feedback throughout our collaborations. Each of them have helped shaping me as a researchers, and I am grateful for the time they have taken to work with me throughout my (as of yet) brief research career. Furthermore, I would like to thank the funding agencies that have (partially) funded my work. Specifically, I wish to thank the danish Council for Independent Research (DFF), Aalborg University's Talent Management Programme, and the

Poul Due Jensen Foundation.

Last, I would like to thank my entire family for all their support during my Ph.D. studies. Their understanding, support, and patience has helped me through what, at times, has been long and odd working hours.

This thesis is written in loving memory of my late father, Brian Aebeloe.

Christian Aebeloe
Aalborg University, Friday 15th July, 2022

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, Aalborg University's Talent Programme, and the Poul Due Jensen Foundation.

Contents

Thesis Details

Thesis Title:	Decentralized Knowledge Graphs on the Web
Ph.D. Student:	Christian Aebeloe
Supervisor:	Professor Katja Hose, Aalborg University
Co-Supervisor	Associate Professor Gabriela Montoya, Aalborg University

The main body of this thesis consists of the following six papers.

[A] Amr Azzam, **Christian Aebeloe**, Gabriela Montoya, Ilkcan Keles, Axel Polleres, Katja Hose, "WiseKG: Balanced Access to Web Knowledge Graphs," *In Proceedings of the 30th The Web Conference (WWW 2021)*, pp. 1422-1434, 2021.

[B] **Christian Aebeloe**, Gabriela Montoya, Katja Hose, "A Decentralized Architecture for Sharing and Querying Semantic Data," *In Proceedings of the 16th Extended Semantic Web Conference (ESWC 2019)*, pp. 3-18, 2019.

[C] **Christian Aebeloe**, Gabriela Montoya, Katja Hose, "Decentralized Indexing over a Network of RDF Peers," *In Proceedings of the 18th International Semantic Web Conference (ISWC 2019)*, pp. 3-20, 2019.

[D] **Christian Aebeloe**, Gabriela Montoya, Katja Hose, "ColChain: Collaborative Linked Data Networks," *In Proceedings of the 30th The Web Conference (WWW 2021)*, pp. 1385-1396, 2021.

[E] **Christian Aebeloe**, Gabriela Montoya, Katja Hose, "A Demonstration of ColChain: Collaborative Knowledge Chains," *In Proceedings of the ISWC 2021 Posters, Demos and Industry Tracks (ISWC 2021)*, 2021.

[F] **Christian Aebeloe**, Gabriela Montoya, Katja Hose, "Optimizing SPARQL Queries over Decentralized Knowledge Graphs," *Unpublished manuscript*.

In addition to the above papers, I have co-authored the following four papers as part of my studies, which are not included in this thesis.

[G] **Christian Aebeloe**, Ilkcan Keles, Gabriela Montoya, Katja Hose, "Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns," *In arXiv Preprint arXiv:2002.09172*, 2020.

[H] Gabriela Montoya, **Christian Aebeloe**, Katja Hose, "Towards Efficient Query Processing over Heterogeneous RDF Interfaces," *In Proceedings of the 2nd Workshop on Decentralizing the Semantic Web (DeSemWeb@ISWC 2018)*, 2018.

[I] **Christian Aebeloe**, Gabriela Montoya, Vinay Setty, Katja Hose, "Discovering Diversified Paths in Knowledge Bases," *In Proceedings of the VLDB Endowment 11 (12)*, pp. 2002-2005, 2018.

[J] **Christian Aebeloe**, Vinay Setty, Gabriela Montoya, Katja Hose, "Top-K Diversification for Path Queries in Knowledge Graphs," *In Proceedings of the ISWC 2018 Posters, Demos and Industry Tracks (ISWC 2018)*, 2018.

This thesis has been submitted for assessment in partial fulfillment of the PhD degree. The thesis is based on the scientific papers that are listed above. Parts of the papers are used directly or indirectly in the extended summary of the thesis. As part of the assessment, co-author statements have been made available to the assessment committee and are also available at the Technical Faculty of IT and Design at Aalborg University. The permission for using the published and accepted articles in the thesis has been obtained from the corresponding publishers with the conditions that they are cited and DOI pointers and/or copyright/credits are placed prominently in the references.

Part I

Thesis Summary

Decentralized Knowledge Graphs on the Web

1 Introduction

This thesis details our effort into making the Web of Data more available, updatable, and scalable through decentralized technologies like Peer-to-Peer (P2P) systems and blockchains. First, Section 1.1 addresses the problem at hand and the motivational aspects of using decentralized technologies to solve it. Then, Section 1.2 describes the contributions of the thesis, followed by Section 1.3 describing the structure of the thesis summary.

1.1 Background and Motivation

The Semantic Web [20] in principle offers access to a vast and interlinked Web of Data [46] made available as Linked Open Data (LOD) [37, 40] using well-known standards such as the Resource Description Framework (RDF) [25] and structured query languages like SPARQL [24]. The promise of such a vast and interlinked Web of Data has, in part, fueled a rapid increase in the number of openly published knowledge graphs on the Web [47] that span a broad range of topics such as life sciences (e.g., Bio2RDF [27]), general knowledge (e.g., DBpedia [17] and Wikidata [85]), geography (e.g., LinkedGeoData [81]), and government data (e.g., US Government LOD [43]). In fact, as of the time of writing, the LOD Cloud [62] consists of 1,301 different knowledge graph with 16,283 links between them. Furthermore, knowledge graphs grow increasingly large; for instance, Wikidata [85] as well as Bio2RDF [27] both contain more than 14 billion triples.

However, the sheer amount of RDF data available today, and the current structure of the Semantic Web, accentuates several major issues that prevent the Web of Data from reaching its full potential. While we addressed one such issue in our previous work [10, 11], i.e., the lack of support for finding paths between nodes in SPARQL 1.1 [24], this thesis addresses the three

major challenges the Semantic Web faces today. First, as we have previously highlighted in [4, 64], data providers have to maintain continuous access to their data; however, as several recent studies [16, 83, 84] have pointed out, this can be quite costly, and with very little monetary incentives to do so, the services become unstable and the data, in the worst case, can become unavailable. Second, data providers have to keep their data up to date which, since current architectures do not support mechanisms to update outdated or faulty data, is often done by periodically publishing a new version of the entire dataset and taking the old version offline. However, accessing previous dataset versions as well as tracing back updates to their origin could be helpful for many applications [3], e.g., to trust data quality. Third, data providers have to ensure efficient and scalable query processing under load, i.e., when a large number of users issue queries at the same time.

In this thesis, we will refer to the aforementioned challenges as the *availability*, *updatability*, and *scalability* challenges, respectively. The availability and scalability challenges often go hand in hand, and several recent approaches have attempted to solve them from different perspectives. For instance, systems based on Linked Data Fragments (LDF) [84] such as [4, 19, 38, 64, 84] aim to increase the availability and scalability of the server in a client-server architecture by shifting parts of the query processing load to the client, increasing the availability of the server. For instance, we recently introduced Star Pattern Fragments (SPF) [4], which processes star-shaped subqueries on the server while processing joins and other SPARQL operators on the client. On the other hand, Peer-to-Peer (P2P) systems [22, 30, 54, 55, 61] address the availability and scalability challenges by replicating the data across multiple nodes in a P2P system and distributing the query processing effort across the network, ensuring that even when the uploader fails, the remaining nodes can still access the data.

However, the vision of a vast Web of Data is still inhibited by the heavy burden on the data providers since existing systems fail to address the challenges completely. In this thesis, we therefore investigate each of the aforementioned challenges with the aim of making knowledge graphs on the Web available, scalable, and updatable, and thus making the Semantic Web one step closer to realizing its full potential. To do this, we first tackle the availability and scalability challenges from two different perspectives; (1) by finding a better balance between putting the load on the server and the client compared to existing client-server architectures, thereby increasing the availability of the server under load, and (2) by replicating the data across a P2P network to ensure data availability even if the uploader fails. Then, we combine the aforementioned P2P architectures with update chains similar to blockchains [33, 69, 80, 86] and let users collaborate on keeping the data up to date, as well as to process queries over earlier versions of the data. Last, we explore how query processing efficiency can be increased when the afore-

mentioned P2P architectures are under heavy load using query optimization strategies based on data locality and cardinality estimation [34, 67, 70].

As such, by building upon state-of-the-art decentralized technologies such as LDFs, P2P systems, and blockchains, we aim to deliver robust methodologies alongside advanced tools and technologies that increase the availability of knowledge graphs while allowing users to collaborate on keeping the data up to date. Furthermore, we aim to provide efficient and scalable query processing techniques over such decentralized knowledge graphs to accommodate for quick and continuous access to the information within the knowledge graphs when a large number of queries are issued concurrently. In the remainder of this section, we give an overview of the contributions of the thesis as well as an overview of the structure of the thesis. Details on the state of the art and the individual contributions are given in Sections 2-7.

1.2 Contributions of the Thesis

The main contributions are novel methodologies and systems that *increase the availability, updatability, and scalability of knowledge graphs on the Web.*

First, Paper A [18] explores load-balancing in client-server architectures to increase the availability and scalability of the server under high load by introducing WiseKG. WiseKG decomposes the query into star-shaped subqueries and determines, based on factors such as the current server load and the network download speed, whether a subquery should be processed by the server or client.

Then, we investigate the three aforementioned challenges from a different standpoint; P2P systems. The objective is to end up with an unstructured P2P system that accommodates data replication, collaborative updates, and scalable query processing under load. This is visualized in Figure 1, where we outlined a vision of a P2P network in which chains of updates allow nodes to collaborate on keeping the data up to date, and distributed indexes allow for scalable query processing under load.

In order to achieve this goal, we propose in Paper B [5] an unstructured architecture for sharing and querying knowledge graphs called Piqnic (a P2p clIent for Query processiNg over semantIC data). Piqnic formally defines an unstructured architecture in which nodes maintain a limited local datastore and a set of neighboring nodes (i.e., a local view over the network) that uses replication to make knowledge graphs more available.

Building upon the architecture of Piqnic, in Paper C [6], we provide a novel scheme for distributed indexes over knowledge graphs in a P2P network that represents the constituents of the knowledge graphs as partitioned bitvectors, removing the need for flooding the network with requests when processing queries.

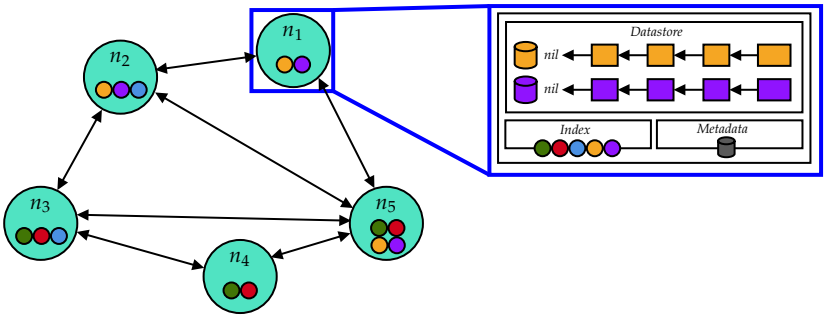


Fig. 1: Example of an unstructured P2P network with update chains and a distributed index.

In Paper D [7], we build on the architectural aspects defined in Papers B [5] and C [6] and propose an approach that enables users to collaborate on keeping the data up to date, called CoLChain (COLlaborative knowledge CHAINs). CoLChain divides the network into *communities* of nodes and uses chains of updates, similar to blockchains, coupled with community-wide consensus to enable users to collaborate on keeping the data up to date, making knowledge graphs updatable. Furthermore, CoLChain lets users roll-back updates and process queries over earlier versions of the data. CoLChain is demonstrated in Paper E [8].

Building upon Piqnic and CoLChain, Paper F [9] introduces a novel approach to optimizing queries in such P2P setups called Lothbrok. Lothbrok uses a query optimization strategy based on star-shaped decomposition that considers the locality and compatibility of knowledge graphs (i.e., whether or not two knowledge graphs may produce join results for a given query).

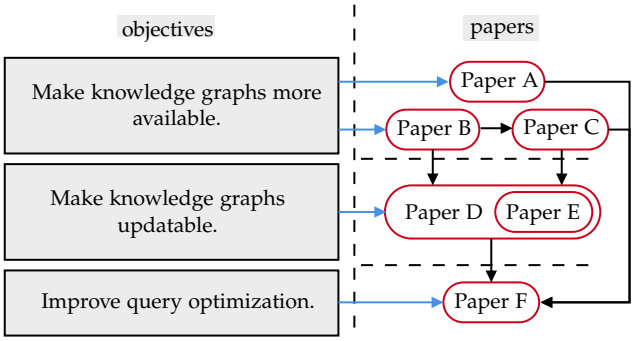


Fig. 2: Overview of the objectives of each paper and connections between papers.

Figure 2 shows which objective each paper in the thesis contributes to along with the connection between the papers. Paper A [18] focus on the availability and scalability issues in a client-server architecture. Papers B [5]

and C [6] then looks into the availability and scalability issues from the P2P perspective. As such, they propose the baseline P2P architecture for sharing and querying knowledge graphs. Furthermore, Paper C builds on Paper B and extends the architecture with the distributed indexes. Papers D [7] and E [8] then focus on making knowledge graphs updatable by extending the architecture defined in Papers B and C using update chains and community-wide consensus to allow users to collaborate on keeping the data up to date. Last, Paper F [9] builds upon the architectures presented in Papers B, C, and D to optimize query processing in P2P architectures.

1.3 Structure of the Thesis

The thesis is structured as follows. Part I contains the motivational aspects of the thesis as well as a summary of the papers included in the thesis. We discuss the related work for decentralized management and querying of knowledge graphs in Section 2. Section 3 summarizes Paper A and our approach to load-balancing in client-server architectures (WiseKG). Then, Sections 4 and 5 summarize our approach to addressing the availability and scalability challenges with unstructured P2P networks (Paper B, Section 4) and decentralized indexes (Paper C, Section 5). We introduce ColChain and summarize how we address the updatability challenge with chains of updates and community-wide consensus in Section 6, which summarizes Papers D [7] and E [8]. Then, Section 7 summarizes Lothbrok [9], i.e., our approach to optimizing queries in the P2P architectures (Paper F). Finally, Section 8 provides a summary of the conclusions and contributions made in this thesis, while Section 9 presents an outlook on future work directions for our work.

Part II reproduces each of the six papers in full with only the layout being revised to fit the format of the thesis. Since some of the papers build upon the work done in previous papers, in accordance with Figure 2, it is recommended to read them in sequential order.

2 State of the Art

SPARQL endpoints are one of the most popular interfaces for storing and querying RDF data today. SPARQL [24] is the most common query language for RDF data today; SPARQL endpoints are Web services that implement the SPARQL protocol. They usually provide an HTTP interface that accepts and answers SPARQL queries [5, 9]. However, as several studies [16, 83, 84] have pointed out, relying on the data providers to maintain endpoints that provide access to their data often results in unavailable and outdated data. To make knowledge graphs more available and up to date, and to provide scalable access to them under high load, this thesis focuses on addressing three of

the main issues the Semantic Web faces today; availability, updatability, and scalability. In this section, we analyze state-of-the-art approaches that address each of these aspects and discuss their advantages and shortcomings.

2.1 Client-Server Architectures

Centralized client-server architectures based on Linked Data Fragments (LDF), such as Triple Pattern Fragments (TPF) [84], Bindings-Restricted TPF (brTPF) [38], Star Pattern Fragments (SPF) [4], and Smart-KG [19], attempt to make the server more available and scalable under high query processing load (i.e., when a lot of queries are executed concurrently) by ensuring that the server only processes requests with low time complexity thus shifting some of the query processing load back to the client. Figure 3 shows an overview of how much of the query processing load the different HTTP interfaces for RDF data put on the server; from downloadable *data dumps* to the left, which requires the clients to process the entire query, to SPARQL endpoints to the right, which process the entire query on the server.

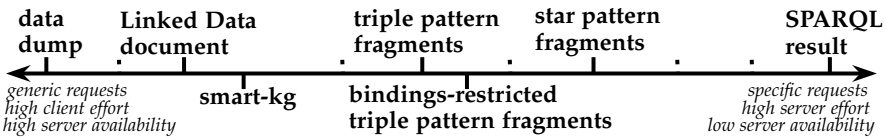


Fig. 3: Overview of different HTTP interfaces for RDF data (adapted from [4, 38, 84]).

As such, each system strikes a different balance between shifting more or less load from the server to the client. For instance, TPF [84] servers only process individual triple patterns, shifting joins and other expensive operations to the clients. Since such requests are usually quite limited in time complexity, and that TPF splits results to such requests into reasonably sized pages of results, the load on the server remains relatively low even when a large number of clients issue queries concurrently, thus increasing the scalability of the server under load. Different TPF clients use different client-side query processing strategies, e.g., based on a greedy query processing algorithm [84], a metadata-based strategy [45], or star-shaped query decomposition coupled with query execution schedulers adapted to data availability and runtime conditions [1]. In any case, however, TPF clients have to transfer bindings from previously evaluated triple patterns along with subsequent triple pattern requests one by one to avoid an unnecessarily large data transfer. Furthermore, [56] proposes a TPF client for processing skyline queries. This naturally incurs a large network overhead, decreasing overall query performance. Additionally, previous studies have found that the performance of TPF is greatly affected by factors such as the shape of the query [65, 66] and

the position of variables in the triple patterns [42].

To decrease the network overhead, brTPF [38] proposed to send bindings from previously evaluated triple patterns in bulks with subsequent requests, thereby lowering the number of requests overall. However, while brTPF indeed decreases the network overhead compared to TPF, complex queries with a large number of intermediate results still incur high network usage. To overcome this limitation, hybridSE [64] combines the brTPF server with a SPARQL endpoint by exploiting the strengths of each interface. In particular, hybridSE processes subqueries with a small number of intermediate results using the brTPF server and sends subqueries with a large number of intermediate results to the SPARQL endpoint. However, since hybridSE sends large subqueries to the SPARQL endpoint, it ends up putting significantly more load on the server while also being vulnerable to server failure.

Instead, SPF [4] proposes to send star-shaped conjunctive subqueries to the server while processing more complex graph patterns on the client. The intuition is that the intermediate results from the triple patterns within each star-shaped subquery that would otherwise have to be transferred to and from the server can be processed internally on the server since such conjunctive subqueries can be processed relatively efficiently by the server [72]. However, such a decomposition strategy based on star-shaped subqueries also shifts a large part of the query processing load to the server, and thus does not utilize the distributed client-side query processing capabilities fully. On the other hand, Smart-KG [19] ships predicate-family partitions based on the star-shaped decomposition of the query to the client and processes the entire query on the client. In opposition to SPF, this shifts the entire query processing load to the client, increasing the scalability and availability of the server, but in doing so incurs an excessive amount of unnecessary data transfer, since the entire partitions are transferred to the client regardless of any bindings obtained from previously evaluated graph patterns, while not utilizing the often stronger server-side resources.

Differently from LDF systems, SaGe [63] uses Web preemption to increase the scalability of the server under load by suspending queries after a fixed time quantum, thereby preventing long-running queries from exhausting the resources made available by the server. After a query has been suspended, it can then be restarted by issuing a new request to the server. However, while as a result, smaller queries are able to complete relatively efficiently even when the server is under load, SaGe still processes the entire, possibly quite complex, queries on the server which, as stated above, is subject to failure.

2.2 Federated Systems

Federated query engines such as the ones presented in [2, 23, 32, 79] enable processing queries over multiple different datasets spread out over multi-

ple SPARQL endpoints or LDF servers [41]. Generally, federated systems have different strategies when processing queries over a federation. For instance, [2] adapts the query execution schedulers to query runtime and the availability of each endpoint in the federation, while [41] processes queries over heterogeneous federated LDF servers by adapting the query optimizer to the characteristics of each interface and the locality of the data.

Commonly for federated systems, however, is that such systems often generate suboptimal query execution plans and join orders, increasing the number of intermediate results and the load on each individual endpoint [52]. To address this issue, extensive research has gone into optimizing the query processing performance of federated systems in different ways [36, 48, 67, 68, 78, 82]. For instance, [68] decomposes the queries into subqueries that can be processed over a single endpoint. On the other hand, [67] relies on the statistical information provided by characteristic sets [70] and pairs [34] combined with Dynamic Programming (DP) to estimate the selectivity of joins and, in doing so, produce more efficient query execution plans. Furthermore, [79] relies on building an index of the data over time by remembering which endpoints provide access to which data.

However, while federated systems generally spread out the query processing load across multiple servers, they target setups where each endpoint or LDF server provides access to different datasets, and thus each individual dataset is still subject to failure.

2.3 Peer-to-Peer Systems

The solutions discussed until now have focused on situations where a single server or a set of servers have attempted to increase the scalability of the data under high load. However, as described earlier, the central points of failure are subject to failure [16, 83] to the detriment of the availability of the data.

To combat the availability problem in particular, Peer-to-Peer (P2P) systems like [22, 30, 54, 55, 61] have recently gained attention in the Semantic Web community. They tackle the availability from a different standpoint; by replicating the data throughout several nodes in a P2P network and completely removing the central point of failure, they ensure the availability of the data even in the event that the original uploader fails. Nodes in P2P systems thus maintain both a local datastore and a local view over the network (i.e., a set of *neighboring* nodes), acting as both server and client.

One recent application of P2P technology on the Web is the Solid [61] project, which proposes to store personal data in so-called decentralized data PODs (Personal Online Datastores) that can be stored on any server decided by the user. As such, data PODs can be scattered across several servers on the Web and, even if a server fails, most users' PODs will still be available. However, Solid in its current form focuses mostly on the protection of personal

data whereas this thesis focuses on the availability of open data.

While the structure of the network and the strategy for replicating and allocating data throughout the network can vary from system to system, most P2P systems today that maintain RDF data provide a structured overlay over the network. For instance, the approaches presented in [22, 54, 55] use Dynamic Hash Tables (DHTs) [57] to determine data locality and allocation. Furthermore, to optimize query processing in such decentralized setups, distributed indexes [26, 36, 82] have been proposed to keep track of which data is located on which nodes.

However, in situations where nodes frequently leave or join the network, or upload new datasets to the network, systems using a structured overlay have to undergo a costly adjustment process when experiencing churn. As such, this thesis proposes, among other contributions, an unstructured P2P network that uses replication to increase data availability while being robust under churn.

2.4 Blockchains

Thus far, we have discussed related approaches proposed to solve the availability and scalability problems. However, all of the approaches discussed until now rely on the data providers to keep the uploaded data up to date. This, however, can often lead to the data becoming outdated or nonfactual [3]. One solution to the updatability problem could be to let users collaborate on keeping the data up to date. To do this while reducing the risk of malicious updates, blockchains could be used.

Blockchains [69] define global ledgers of data that all the nodes within the network store, i.e., chains of data blocks. They rely on the consensus (majority agreement) on the state of the ledger, to ensure that data pushed to the ledger is factual. However, using blockchains in the context of decentralized knowledge graphs has, to the best of our knowledge, only been briefly researched previously [33, 80]. Furthermore, blockchains require all the participating nodes to store the entire ledger, pack the structured data into blocks of a fixed size, and guarantee the immutability of the data [86].

In the database community, blockchains have previously been used for the storage and retrieval of relational data [14, 28]. For instance, CAPER [14] proposes to store the ledger as a directed acyclic graph such that each node only has to store a limited part of the ledger. On the other hand, BlockchainDB [28] provides a partitioned relational database layer on top of an existing blockchain. In any case, however, such systems either limit the autonomy of each node, still require all the nodes to store the entire ledger, enforce shared relational schema on top of the blockchain on each node, or view the data within the blockchain as one big dataset instead of multiple datasets owned by different providers.

3 Scalable Access to Knowledge Graphs

This section gives an overview of Paper A [18].

3.1 Motivation and Problem Statement

As discussed in Section 1.1, the increasing amount of RDF data published on the Web today puts an ever increasing burden on the data providers to maintain access to the data. As a result, many publicly available knowledge graphs often become unavailable [16, 83] and the data inaccessible. And while solutions based on Linked Data Fragments (LDF) [4, 19, 38, 84] do decrease the load on the server by shifting some of the query processing load back to the client, they either suffer from a high network overhead [38, 84] or put the bulk of the query processing load on either the clients [19] or the server [4], suffering from imbalanced load. As such, in Paper A [18], we present WiseKG that dynamically attempts to balance the query processing load between the server and client. In particular, we combine two state-of-the-art LDF interfaces that decompose the queries into star-shaped subqueries that offer different balances between network usage and server load and take advantage of the strengths of each interface to provide more efficient query processing performance under load. WiseKG proposes a cost model that determines at query time whether it is most efficient to process a particular star-shaped subquery on the server or the client based on factors such as the current load on the server, data transfer overhead, and the resources available on the client. In doing so, WiseKG increases scalability and availability when the server is under heavy load.

3.2 Preliminaries

The Resource Description Framework (RDF) [25] is a commonly used format for storing semantic data. RDF structures the data as triples that we formally define as follows:

Definition 1 (RDF Triple – adapted from [5, 7, 18])
Given I, B, and L, i.e., the disjoint sets of IRIs, blank nodes, and literals, an RDF triple t is of the form $t = (s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where s, p, and o are called subject, predicate, and object.

As in [5, 7, 18], we define a *knowledge graph* $\mathcal{G}$ as a finite set of RDF triples. A SPARQL query [24] is of one or more Basic Graph Patterns (BGPs) combined with operations such as `UNION` and `OPTIONAL`. A BGP is a set of conjunctive *triple patterns*. Given again the sets I, B, and L, as well as the set V of all variables disjoint with I, B, and L, a triple pattern tp is a triple of the form $tp = (s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$. Given a

triple pattern tp, we denote $subj(tp)$, $pred(tp)$, and $obj(tp)$ as the subject, predicate, and object of tp, respectively. Furthermore, we denote the left-linear execution order for a BGP $P = \{tp_1, \ldots, tp_n\}$ as a sequence, i.e., the execution plan $(tp_1, \ldots, tp_2)$ is equivalent to the execution plan denoted by $(((tp_1 \bowtie tp_2) \bowtie \ldots) \bowtie tp_n)$. The answer to a BGP P over a knowledge graph $\mathcal{G}$, denoted $[[P]]_{\mathcal{G}}$ is a set Ω of *solution mappings*, defined formally in [18] as:

Definition 2 (Solution mapping [84] – adapted from [5, 7, 18])
Given a BGP P and a knowledge graph $\mathcal{G}$, I, B, L are the sets of IRIs, blank nodes, and literals in $\mathcal{G}$, and V is the set of variables in P. A *solution mapping* μ is a partial mapping $\mu : V \mapsto (I \cup B \cup L)$.

The following description is adapted from [5, 7, 18]. Given a triple pattern tp and a solution mapping μ, $\mu[tp]$ denotes the triple (pattern) obtained by replacing the variables in tp with the values specified in μ. A triple t is a *matching* triple to a triple pattern tp iff there exists a solution mapping μ for which $t = \mu[tp]$. Moreover, $dom(\mu)$ denotes the *domain* of a solution mapping μ, i.e., the set of bound variables in μ, and $vars(tp)$ denotes the variables in a triple pattern tp. Last, two solution mappings μ_1, μ_2 are said to be *compatible*, denoted $\mu_1 || \mu_2$, iff for any $v \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(v) = \mu_2(v)$.

A *star pattern* sp is a BGP such that $sp = \{tp_1, \ldots, tp_k\}$ and $\forall tp_i = (s_i, p_i, o_i) \in sp$ and $\forall tp_k = (s_k, p_k, o_k) \in sp$, $s_i = s_k$, i.e., all triple patterns share the same subject. Given a BGP P, we define the *star decomposition* of P:

Definition 3 (Star decomposition – adapted from [18])
Given a BGP P, the *star decomposition* of P, denoted $\mathcal{S}(P)$, is defined as:

$$\mathcal{S}(P) = \{\{tp \in P \mid subj(tp) = s\} \mid s \in subj(P)\}$$

Linked Data Fragments

In the following, we formally define LDF APIs [84] by reproducing the definitions and formulas from [18]. As in [18], we omit some details on LDFs like metadata that is sent along with query results, as well as hypermedia results. As such, we define an LDF API as follows:

Definition 4 (Linked Data Fragment – adapted from [18])
Given a knowledge graph $\mathcal{G}$, An LDF API f of $\mathcal{G}$ accessible at an endpoint IRI i is a tuple $f = (s, \Phi)$ where:

- s is a selector function $s(\mathcal{G}, P, \Omega)$ that defines how a fragment $\Gamma \subseteq \mathcal{G}$, or alternatively a set of fragments $\Gamma^* \subseteq 2^{\mathcal{G}}$, is constructed for a given BGP P and set of solution mappings Ω.
- Φ is a paging mechanism $\Phi(n, l, o)$ parameterized by $n, l, o \in \mathbf{N}_0$ denoting maximum page size, limit, and offset.

We define two variants of the selector function, $s(\mathcal{G}, P, \Omega)$ and $s^*(\mathcal{G}, P, \Omega)$, which differ in returning either a single (sub)graph or one (sub)graph per solution $\mu \in [[P]]_\mathcal{G}$. These are reproduced from [18] and defined as follows:

$$s(\mathcal{G}, P, \Omega) = \{t \in \mu[P] \mid \exists \mu \in [[P]]_\mathcal{G} : \mathcal{G} \models \mu[P] \wedge \exists \mu' \in \Omega : \mu || \mu'\} \quad (1)$$

$$s^*(\mathcal{G}, P, \Omega) = \{\mu[P] \mid \exists \mu \in [[P]]_\mathcal{G} : \mathcal{G} \models \mu[P] \wedge \exists \mu' \in \Omega : \mu || \mu'\} \quad (2)$$

In the remainder of this section, we will describe four different existing LDF APIs in terms of the above definitions and formulas by reproducing them from [18]: Triple Pattern Fragments (TPF) [84], Bindings-Restricted TPF (brTPF) [38], Star Pattern Fragments (SPF) [4], and Smart-KG [19].

Triple Pattern Fragments (TPF) [84] can be described using the generic LDF framework from Definition 4 as follows:

- The selector function is $s(\mathcal{G}, P, \Omega)$ as shown in Equation 1
- The only admissible form of P are triple patterns and $\Omega = \{\emptyset\}$
- $\Phi(n, l, o)$ batches results into pages of n triples; l and o cannot be set explicitly.

Bindigs-Restricted TPF (brTPF) [38] is an extension of TPF, and the description as an LDF system is similar to the one for TPF as well. The difference being, that brTPF allows arbitrary $\Omega \neq \{\emptyset\}$.

Star Pattern Fragments (SPF) [4] generalizes brTPF to handle star patterns rather than triple patterns. As such, SPF is described as an instance of an LDF API as follows:

- The selector function is $s^*(\mathcal{G}, P, \Omega)$ as shown in Equation 2
- The only admissible form of P are star patterns
- Ω can be any set of bindings
- $\Phi(n, l, o)$: n is fixed to P of size k but SPF allows to retrieve chunks of l results (iterating over increasing offsets $o = o + l$)

Smart-KG [19] ships knowledge graph partitions to the client per star pattern and processes the query on the client. As such, the server holds partitions per *characteristic set* [70] of a knowledge graph $\mathcal{G}$. A characteristic set is a set of predicates and is, for any subject in $\mathcal{G}$, defined as follows:

Definition 5 (Characteristic set – adapted from [18])
Given a knowledge graph $\mathcal{G}$, the *characteristic set* of a subject s over $\mathcal{G}$, $C_\mathcal{G}(s)$, is the set of predicates associated with s:

$$C_\mathcal{G}(s) = \{p \mid \exists o : (s, p, o) \in \mathcal{G}\} \quad (3)$$

We denote the set of characteristic sets of a knowledge graph $\mathcal{G}$ as $C(\mathcal{G})$ or C for simplicity whereby $C(\mathcal{G}) = \{C_\mathcal{G}(s) | s \in subj(\mathcal{G})\}$.

Given the definition of a characteristic set, Smart-KG can be described as an LDF interface as follows:

- The selector function is adapted from Equation 1 as follows:

$$s_{SKG}(\mathcal{G}, P, \Omega) = \{t \in \mathcal{G} \mid \exists t' \in s(\mathcal{G}, P, \Omega) : subj(t) = subj(t')\} \quad (4)$$

- Admissible patterns are defined by submitting a characteristic set $c = \{p_1, \ldots, p_k\}$ that can be interpreted as a set of triple patterns $\cup_{i=1}^{k}\{(?s, p_i, ?o_i)\}$, and $\Omega = \{\emptyset\}$
- Φ: Only $n = \infty$ is admissible, i.e., no pagination is supported since the union of all relevant partitions is returned

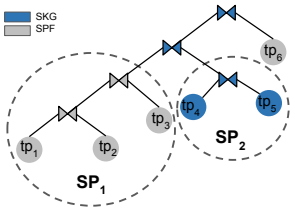
3.3 WiseKG: Balanced Access to Web Knowledge Graphs

All the LDF interfaces mentioned so far suffer from imbalanced load either towards the client (TPF, brTPF, Smart-KG) or the server (SPF). To this end, we present WiseKG, a system that attempts to optimally distribute the processing of star-shaped subqueries to take advantage of the strengths of SPF and Smart-KG in particular. In other words, WiseKG dynamically determines whether it is more efficient to process a particular star pattern on the client or server.

Example 1 (Motivating example – adapted from [18])
Consider query Q shown in Figure 4a over DBpedia [17]. All triple patterns in the query have large cardinalities (the smallest cardinality being 146,716), meaning both TPF and brTPF would have to send an excessive number of requests to process the query, and transfer a large number of intermediate results back to the client. The star decomposition of the query results in two star patterns, $sp_1 = \{tp_1, tp_2, tp_3\}$ and $sp_2 = \{tp_4, tp_5\}$, and tp_6 which is a star pattern with a single triple pattern. sp_1 has 89,366 solutions and sp_2 has 600,349 solutions. Both SPF and Smart-KG order the query in the following join order: (sp_1, sp_2, tp_6). When processing sp_1, Smart-KG then ships a partition with 1,628,572 distinct stars, whereas SPF only ships the 89,366 resulting bindings, resulting in less of a network overhead. However, since SPF enforces a limit on the number of bindings attached to each request, when processing the join between sp_1 and sp_2, SPF has to bulk the bindings from sp_1 into batches of, for instance, 30 bindings to transfer along with the request, amounting to $2,979$ requests in total. We could mitigate this overhead by instead shipping the compressed partition with the 600,349 solution mappings to sp_2 and processing the join locally on the client, illustrating a situation where it might be a good idea to process parts of the query on the server, but other parts on the client.

```
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 (Q)



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

Fig. 4: Example of processing a SPARQL query with WiseKG (reproduced from [18]).

Overview

WiseKG uses a cost model to determine whether it is most efficient to process a star-shaped subquery on the client or on the server, i.e., using SPF or SmartKG. To do this, it creates a query execution plan *annotated* with the approach to use for each subquery, denoting annotations as *SPF* in the case of server-side processing, and *SKG* in the case of client-side processing. For instance, the plan $\Pi = (sp_1^{SPF}, sp_2^{SKG}, tp_6^{SPF})$ denotes the execution plan visualized in Figure 4b and denotes that sp_1 and tp_6 are processed by the server and sp_2 by the client.

Upon receiving a query, the WiseKG server decomposes it into star patterns, uses its cost model to obtain an annotated query execution plan, and attaches to it a timestamp τ that denotes the expiration time of the plan; in case the client has not finished execution by τ, it has to send a new request to obtain a new query execution plan which may look different. This is to ensure the server resources are always optimally utilized. Formally, the WiseKG API offers the following interfaces to access a knowledge graph $\mathcal{G}$ [18]:

- A control $SPF(P, \Omega)$ returning $s^*(\mathcal{G}, P, \Omega)$ (Equation 2)
- A control $SKG(P, \Omega)$ returning $s_{SKG}(\mathcal{G}, P, \emptyset)$ (Equation 4)
- An execution plan interface $Plan(P)$ returning a *pair* (Π_P, τ)

Given a control $c \in \{SPF, SKG\}$, $c(P, \Omega)$ denotes that P is executed by calling either $SPF(P, \Omega)$ or $SKG(P, \Omega)$ acording to c [18]. [18] assumes that the call to $c(P, \Omega)$ is converted to a set of bindings through the function $eval_c(P, \Omega) = \Omega \bowtie [[P]]_{\mathcal{G}}$. Furthermore, the call to $Plan(P)$ attaches to the execution plan a timestamp $\tau = \tau_{\mathcal{C}} + \iota$ where $\tau_{\mathcal{C}}$ is the current timestamp and ι is a fixed time quantum similar to SaGe's [63].

Cost Model

The cost model used by WiseKG [18] is inspired by the R* optimizer [60] from the field of distributed databases [87]. Given a star pattern sp, we use

the following components when estimating the cost of processing sp: estimated number of CPU instructions (#CPU), estimated number of requests to the server (#M), estimated number of transferred bytes (#BYT), and estimated number of I/O operations (#IO), formalized as follows (reproduced from [18]):

$$cost(sp) = \underbrace{W_{CPU} \times (\#CPU)}_{\text{Processing}} + \underbrace{W_M \times (\#M)}_{\text{Messaging}} + \underbrace{W_{BYT} \times (\#BYT)}_{\text{Data transfer}} + \underbrace{W_{IO} \times (\#IO)}_{\text{I/O}}$$

W_{CPU}, W_M, W_{BYT}, and W_{IO} describe weights on each parameter that are determined as follows [18]:

- W_{CPU}: The inverse of the CPU's IPS (Instructions Per Second) rate dampened by the current load on the CPU:

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

- W_M: The average time to transmit a request; in our implementation and experiments, we assume a constant value of $W_M = 50ms$.
- W_{BYT}: The conservative minimum between the bandwidth of the server bw_{server} and client bw_{client}:

$$W_{BYT} = \frac{1}{min(bw_{server}, bw_{client})}$$

- W_{IO}: The time it takes to load 1 MB of data into memory. In our experiments, I/O was a negligible factor (cf. our experiments in Paper A), and we therefore let $W_{IO} = 0$ in our implementation.

The above formulas can be concretized to estimate the cardinality of processing a star pattern respectively on the server, $cost_{SPF}$, and the client, $cost_{SKG}$. To do this, we will first reproduce from [18] the following functions and variables:

- $card(sp, \Pi)$: The estimated cardinality of processing sp using the estimated number of bindings from Π (as described in [70])
- $size(c)$: The size of the partition for a characteristic set c on disk (i.e., the size of the corresponding HDT [29] file)
- i_t: The number of CPU instructions needed to process each triple in the result
- b_t: The (average) number of bytes per triple in the result

Given the above definitions, we reproduce the formulas for the cost estimations from [18]. First, the cost of processing the star pattern on the server can be estimated using the following formula [18]:

$$cost_{SPF}(sp,\Pi) = W_{CPU_{serv}} \times \underbrace{card(sp,\Pi) \times i_t}_{\text{\#CPU}} + \\ W_{MSG} \times \underbrace{\frac{card(sp,\Pi)}{\Phi(n)}}_{\text{\#M}} + W_{BYT} \times \underbrace{card(sp,\Pi) \times b_t}_{\text{\#BYT}} + W_{IO} \times \underbrace{size(\mathcal{G})}_{\text{\#IO}} \tag{5}$$

In other words, the CPU cost is the estimated cardinality multiplied by the number of instructions per results, the messaging cost is the estimated cardinality divided by the page size, and the data transfer cost is the estimated cardinality multiplied with the number of bytes per triple. The cost of shipping the entire partition to the client and processing the star pattern locally is estimated using the following formula [18]:

$$cost_{SKG}(sp,\Pi) = W_{CPU_{client}} \times \underbrace{card(sp,\Pi) \times i_t}_{\text{\#CPU}} + W_{MSG} \times \underbrace{|C_{sp}|}_{\text{\#M}} + \\ W_{BYT} \times \underbrace{\left(\sum_{c \in C_{sp}} size(c) \right)}_{\text{\#BYT}} + W_{I/O} \times \underbrace{\left(\sum_{c \in C_{sp}} size(c) \right)}_{\text{\#IO}} \tag{6}$$

Differently from $cost_{SPF}$, $cost_{SKG}$ estimates the messaging cost as the number of characteristic sets for sp and the data transfer cost as the summarized number of subjects in each relevant characteristic set.

3.4 Query Processing

When processing a query, the WiseKG client follows the following steps (adapted from [18]):

1. Retrieve (Π, τ) by calling $Plan(P)$ on the server
2. For each star pattern $sp^c \in \Pi$ and solution mapping Ω from evaluating previous star patterns, do the following:
   (a) If $\tau < \tau_{\mathcal{C}}$, call $Plan(P)$ once again
   (b) Otherwise, call $c(sp, \Omega)$ on the server

Algorithm 1 shows how the server creates a query execution plan given a BGP P. First, line 2 decomposes P into star patterns. Then, lines 5-13 iteratively find and select the star pattern sp_i with the lowest estimated cardinality and append it to the execution plan. The algorithm estimates the cost of processing sp_i on the server ($cost_{SPF}$, Equation 5) and on the client ($cost_{SKG}$, Equation 6); if $cost_{SPF} \leq cost_{SKG}$, sp_i is appended on line 10, otherwise it is appended on line 12. Finally, the timestamp τ is found on line 14, and (Π_P, τ) is returned on line 15.

Algorithm 1 Create annotated execution plan – reproduced from [18])

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: $S \leftarrow \mathcal{S}(P)$
3: $\Pi_p \leftarrow ()$
4: **while** $S \neq \emptyset$ **do**
5: **for** $sp \in S$ **do**
6: $cnt_{sp} \leftarrow card(sp, \Pi_p)$
7: **if** $cnt_{sp} = 0$ **then return** $()$
8: $sp_i \leftarrow sp$ where $sp \in S$ and $cnt_{sp} \leq cnt_{sp'}$ for all $sp' \in S$
9: **if** $cost_{SPF}(sp_i, \Pi_P) \leq cost_{SKG}(sp_i, \Pi_P)$ **then**
10: $\Pi_P \leftarrow append(\Pi_P, (sp_i^{SPF}))$
11: **else**
12: $\Pi_P \leftarrow append(\Pi_P, (sp_i^{SKG}))$
13: $S \leftarrow S \setminus \{sp_i\}$
14: $\tau \leftarrow \tau_{\mathcal{C}} + \iota$
15: **return** (Π_P, τ)

Algorithm 2 defines a recursive function that processes a query execution plan Π with a timestamp τ. First, line 2 checks whether the timestamp has expired; if it has, the algorithm collects a new plan and expiry timestamp from the server on line 3. Then, if Π consists of a single star pattern, the algorithm processes that star pattern using its control (line 5). Otherwise, the function makes a recursive call for the left subtree (line 7) and uses the resulting bindings in another recursive call for the right subtree (line 8). Last, the resulting bindings are returned on line 9.

3.5 Evaluation and Discussion

To make knowledge graphs published on the Web available and scalable under heavy query processing load, existing LDF systems shift some of the query processing load back to the client in an attempt to decrease the load on the server. However, such systems usually suffer from imbalanced load, either putting too much effort on the server [4, 63] or the client [19, 38, 84]. As such, we argue that there is a middle ground where the load can be balanced across the server and client in such a way that exploits the stronger server-side query processing capabilities without overloading it. To this end, in Paper A [18], we proposed WiseKG; an LDF system that combines two existing approaches [4, 19] and dynamically determines whether each star-shaped subquery in the query can be answered most efficiently by the client

Algorithm 2 Processing a query execution plan – reproduced from [18])

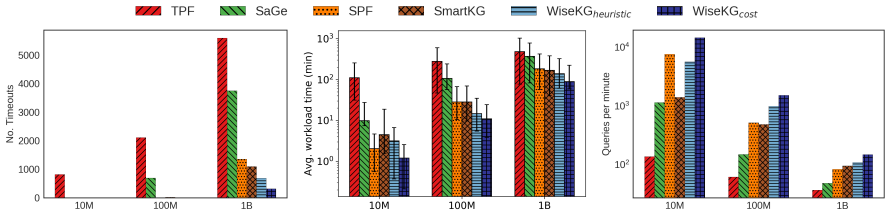
Input: $\Pi = (sp_1^{c_1}, \ldots, sp_n^{c_n})$ // an execution plan;
τ // expiry timestamp;
Ω' // a set of bindings
Output: Ω // set of solution bindings

1: **function** *evalPlan*(Π, τ, Ω')
2: **if** $\tau < \tau_C$ **then**
3: $(\Pi, \tau) \leftarrow Plan(BGP(\Pi))$
4: **if** $\Pi = sp^c$ **then**
5: $\Omega \leftarrow eval_c(sp, \Omega')$
6: **else**
7: $\Omega \leftarrow evalPlan((sp_1^{c_1}, \ldots, sp_{n-1}^{c_{n-1}}), \tau, \Omega')$
8: $\Omega \leftarrow evalPlan(sp_n^{c_n}, \tau, \Omega)$
9: **return** Ω

or the server based on factors such as the current load on the server and the resources available on the client.

To test our approach, we conducted experiments using the WatDiv benchmark tool [13] and generated datasets with 10 million (`watdiv10M`), 100 million (`watdiv100M`), and 1 billion (`watdiv1B`) triples. For queries, we used the query templates and query generator provided by WatDiv to create two query loads using the basic testing templates (`watdiv-btt`) and the stress-testing templates (`watdiv-sts`). Furthermore, we tested our approach using DBpedia [17] (v.2015A) and generated 28 queries from a real-world query log called LSQ [76]. We compared our approach to TPF [84], SaGe [63], SPF [4], and Smart-KG [19]. Furthermore, we included two versions of WiseKG in our experiments; (1) WiseKG$_{heuristic}$ which executes all star patterns on the server until the CPU load reaches a threshold σ, after which the remaining star patterns are processed by the client, and (2) WiseKG$_{cost}$ which uses the cost model defined in Section 3.3. Our full experimental evaluation is given in Paper A [18]; in the following, we will discuss the most interesting results.

Figure 5 shows the number of timeouts, average workload time, and throughput (queries/minute) incurred by each system when 128 clients issue queries concurrently. Clearly WiseKG has better performance than the competing systems when the server is under load. In fact, for `watdiv10M` and `watdiv100M`, WiseKG experiences no timeouts, and even for `watdiv1B`, only around 2% of the queries timed out compared to 13% for Smart-KG, 21% for SPF, and up to 55% for SaGe and TPF (Figure 5a). This is in line with the performance of the systems; WiseKG has a sizable increase in performance, both in terms of the workload time (Figure 5b) and query throughput (Figure 5c) compare to every other system. In fact, WiseKG increases performance by



(a) Number of Timeouts **(b)** Avg. Workload Time **(c)** Queries per minute

Fig. 5: Number of timeouts, average workload time, and throughput for 128 clients over each WatDiv dataset for `watdiv-sts` (reproduced from [18]).

up to two orders of magnitude compared to TPF, and up to an order of magnitude compared to the remaining approaches.

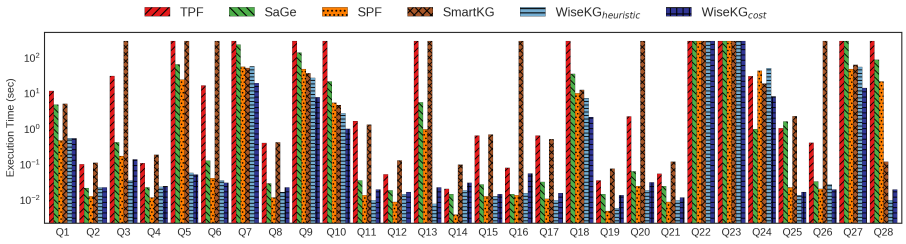
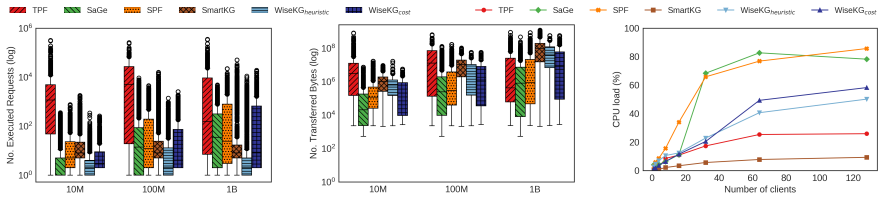


Fig. 6: Execution time (in seconds) for the DBpedia queries (reproduced from [18]).

These findings are supported by Figure 6 which shows the execution time for each system over the DBpedia queries and dataset. Again, WiseKG generally has significantly better performance than the competitors. The only cases where SPF or SaGe is faster than WiseKG are the queries where the overhead of computing the query execution plan is a sizable part of the time complexity (i.e., very simple queries); in these cases, WiseKG$_{heuristic}$ is also faster than WiseKG$_{cost}$ for the same reason.

Figures 7a and 7b show the number of requests each system made to the server and the number of bytes each system transferred when processing the `watdiv-sts` queries over each WatDiv dataset. Clearly, SaGe incurs the lowest network overhead; this is because it does not transfer any intermediate results over the network. On the other hand, while WiseKG issues less requests than SPF, it also has a slightly higher data transfer. This is due to the transfer of some partitions during query processing when the load on the server is high. This is also reflected on Figure 7c which shows the average CPU usage for an increasing number of nodes; the SaGe and SPF servers suffer from high server load which inhibits performance. For WiseKG, though, the CPU load on the server never goes above 60%, showing that while the network overhead might



(a) No. requests to the server for 128 clients over each WatDiv dataset (log).

(b) No. transferred bytes for 128 clients over each WatDiv dataset (log).

(c) Avg. Server CPU Usage (in percentage) for increasing numbers of clients over `watdiv1B`.

Fig. 7: Number of requests to the server and number of transferred bytes for 128 clients over each WatDiv dataset, and CPU load for increasing numbers of clients over `watdiv1B` on the `watdiv-sts` workload (reproduced from [18]).

be a little higher compared to systems like SPF, this is actually preferable, since the server load is maintained on an acceptable level that increases the performance when a lot of clients issue queries concurrently.

Overall, our experimental results show that WiseKG significantly outperforms state-of-the-art LDF interfaces on high demanding workloads and when the server is under load. Furthermore, they show that the cost model used by WiseKG improves the average workload completion time while being able to answer more queries without timing out. However, while our experiments show that WiseKG is able to increase the server availability when it is under heavy load, it still relies on the data provider to maintain a server. As discussed previously, centralized servers are subject to failure [16, 83] and represent a heavy burden on the data providers. As such, in Section 4, we will investigate how such a central point of failure can be removed by replicating data throughout a Peer-to-Peer (P2P) network such that even if the original uploader fails, the data is still available.

4 An Unstructured Peer-to-Peer Architecture

This section gives an overview of Paper B [5].

4.1 Motivation and Problem Statement

While the approach presented in Paper A [18] as well as other LDF approaches [4, 19, 38, 84] do decrease the load on the server by shifting some of the query processing load back to the client, they fail to address the core of the availability problem, i.e., the centralized point of failure that the server represents. As such, removing the central point of failure and using replication is pivotal in ensuring the continued availability of the data even when the original uploader fails. However, the few Peer-to-Peer (P2P) systems that

exist for RDF data [22, 54, 55, 61] enforce a structured overlay over the network that has to be recomputed during churn (when nodes leave or join the network, or when data is added, removed, or modified). As such, Paper B [5] presents PIQNIC (a P2p clIent for Query processiNg over semantIC data), a system that aims to address the availability problem by replicating the data on several nodes throughout a P2P network while enforcing no structured overlay over the network, thereby ensuring node autonomy and being robust under churn.

4.2 PIQNIC: A Decentralized Architecture for Sharing and Querying Semantic Data

PIQNIC builds upon the principles unstructured P2P networks. As such, a PIQNIC network consists of a set of interconnected nodes that each maintain a local datastore and a set of neighboring nodes. The partial overview over the network, i.e., the set of neighboring nodes, consists of (1) nodes with highly related knowledge graphs, i.e., knowledge graphs that use common IRIs, to ensure efficient query processing over multiple knowledge graphs, and (2) some random neighbors to ensure connectivity within the network [5].

Data Fragmentation

Due to the size of knowledge graphs today, replicating entire RDF datasets at several nodes is not always feasible or useful. Hence, PIQNIC proposes a customizable fragmentation approach presented in the following using the definitions and examples reproduced with modifications from Paper B [5]. Given a PIQNIC network N, we formally define a fragment as follows:

Definition 6 (Fragment – adapted from [5])
Let $\mathcal{G}_N$ be a knowledge graph that includes all RDF triples in the PIQNIC network N. A fragment f is a 4-tuple $f = (T, N, u, i)$ where:

- T is a finite set of RDF triples, and $T \subseteq \mathcal{G}_N$,
- N is a set of PIQNIC nodes storing the fragment,
- u is an IRI that identifies the fragment, and
- i is an identification function that determines whether the fragment contains triples matching a given triple pattern.

Identification functions are used as part of the query processing algorithm to determine which fragments contain relevant data to which triple patterns. Since any node can upload a knowledge graph, we include the notion of fragment *ownership*, i.e., the owner of a fragment manages the replication and allocation of the fragment. As such, we formally define a dataset as:

Definition 7 (Dataset – reproduced from [5])
A dataset D is a triple $D = (F, u, o)$ where:

- F is a set of fragments,
- u is an IRI that identifies the dataset, and
- o is an identifier of the "owner" node, i.e., the node that uploaded F to the network.

Given a knowledge graph $\mathcal{G}$, the fragments of $\mathcal{G}$ are obtained using a *fragmentation function* $\mathcal{F}$, formally defined as follows:

Definition 8 (Fragmentation function – reproduced from [5])
A fragmentation function $\mathcal{F}$ is a function that, when applied to a knowledge graph $\mathcal{G}$, creates a set of fragments $F = \mathcal{F}(\mathcal{G})$, i.e., $\mathcal{F}(\mathcal{G}) : \mathcal{G} \mapsto 2^{\mathcal{G}}$.

Fragmentation functions can have different granularities; for instance, the fragmentation function $\mathcal{F}_C(\mathcal{G}) = \{\mathcal{G}\}$ that results in the original knowledge graph is very coarse-granular. For simplicity, PIQNIC uses the predicate-based fragmentation presented in Definition 9; however, in Section 7, we discuss using a more complex fragmentation function based on characteristic sets [70].

Definition 9 (Predicate-based fragmentation function – reproduced from [5])
Let p_t denote the predicate of a triple t. A predicate-based fragmentation function $\mathcal{F}_P(\mathcal{G}) = \{F_p \mid \exists t \in \mathcal{G} : p_t = p \wedge (\forall t' \in \mathcal{G})[p_{t'} = p] : t' \in F_p\}$ defines one fragment for each unique predicate in the knowledge graph $\mathcal{G}$.

Table 1: Applying $\mathcal{F}_P$ to a knowledge graph $\mathcal{G}_E$ (adapted from [5])

(a) Knowledge graph $\mathcal{G}_E$ [5]

Knowledge graph $\mathcal{G}_E$		
(a p_1 b)	(a p_2 c)	(a p_3 d)
(b p_2 e)	(b p_1 d)	(b p_3 d)
(c p_3 d)	(d p_1 c)	(c p_2 a)
(f p_4 d)	(d p_4 f)	(e p_5 g)

(b) $\mathcal{F}_P(\mathcal{G}_E)$ [5]

$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$
(a p_1 b)	(a p_2 c)	(a p_3 d)
(b p_1 d)	(b p_2 e)	(b p_3 d)
(d p_1 c)	(c p_2 a)	(c p_3 d)

$\mathbf{f_4}$	$\mathbf{f_5}$	
(f p_4 d)	(e p_5 g)	
(d p_4 f)		

Example 2 (Fragmentation – reproduced from [5])
Consider the knowledge graph $\mathcal{G}_E$ in Table 1a. Applying $\mathcal{F}_P$ to $\mathcal{G}_E$ results in the set of fragments f_1, f_2, f_3, f_4, and f_5 shown in Table 1b; one fragment for each unique predicate p_1, p_2, p_3, p_4, and p_5.

Network Architecture

We now formally define the network structure of a PIQNIC network using the definitions, equations, and examples adapted from Paper B [5]. A PIQNIC network is a set of interconnected nodes. A node is formally defined as follows:

Definition 10 (Node – reproduced from [5])
A node n is a triple $n = (\Gamma, \Delta, N)$ where:

- Γ is the set of fragments located on the node,
- Δ is a set of datasets owned by the node, and
- N is a set of so-called neighbor nodes in the network.

To account for changes in the network, nodes periodically *shuffle* their neighbors with another node. To do this, a node n picks a random node n' in its set of neighbors ($n' \in n.N$) and selects a subset of its neighbors which it sends to n' removing them from the set $n.N$. The selected neighbors are the ones with the least related data in the local datastore, based on the *joinability* of the fragments within the nodes. Two fragments are said to be joinable if they have some subject/object IRIs in common; formally, the joinability of fragments are defined as follows:

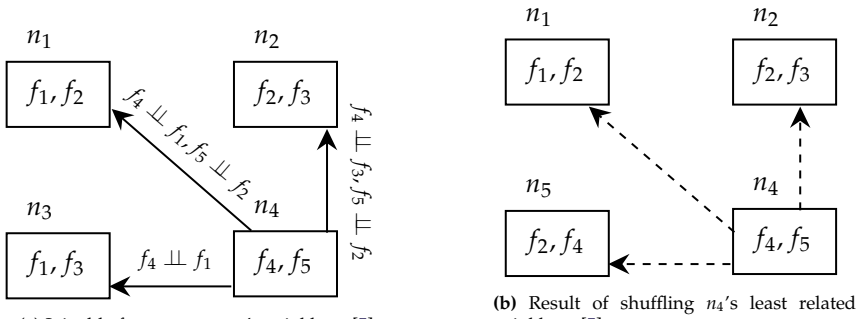
Definition 11 (Fragment joinability – reproduced from [5])
Let s_t and o_t be the subject and object of a triple t, $\mathcal{G}_N$ the knowledge graph containing all RDF triples in a network N, and $f_1, f_2 \in \mathcal{F}(\mathcal{G}_N)$. f_1 and f_2 are said to be "joinable", denoted $f_1 \perp\!\!\!\perp f_2$, iff for at least one triple $t_1 \in f_1$, there exists a triple $t_2 \in f_2$, s.t. $\{s_{t_1}, o_{t_1}\} \cap \{s_{t_2}, o_{t_2}\} \neq \emptyset$.

Notice that Definition 11 does not consider the rate of overlap between two fragments, but only *if* two fragments overlap; this is the case since fragments with a small rate of overlap might still be important for complete query results [5]. We now reproduce from [5] the definition of a relatedness metric to rank a node's neighbors; given a node n, the following function selects the k nodes $R \subseteq n.N$ that minimize the objective function in Equation 7, as follows [5]:

$$Rel(n) = \underset{R \subseteq n.N}{\arg\min} \sum_{n_i \in R} \frac{|Join(n, n_i)|}{|n.\Gamma|} \qquad s.t.\ |R| = k \tag{7}$$

where $Join(n, n_i)$, as defined in Equation 8 and [5], returns the set of fragments stored in n's local datastore that are joinable with at least one of node n_i's fragments with a different fragment identifier, reproduced from [5] as:

$$Join(n_1, n_2) = \{f_1 \in n_1.\Gamma \mid \exists f_2 \in n_2.\Gamma : f_1 \perp\!\!\!\perp f_2 \wedge f_1.u \neq f_2.u\} \tag{8}$$



(a) Joinable fragments at n_4's neighbors [5]

(b) Result of shuffling n_4's least related neighbors [5]

Fig. 8: Computing the relatedness of n_4's neighbors and shuffling. *Solid arrows denote a connection to a neighbor in list n_4.N, and dashed arrows neighbors after a shuffle (reproduced from [5]).*

Example 3 (Neighbor ranking – reproduced from [5])
Given the fragments in Table 1b and the distribution of the fragments between the 4 nodes in Figure 8a, n_4 selects the least related neighbor to shuffle. As such, we apply Equation 7 and obtain:

- n_1: Since $f_4 \perp\!\!\!\perp f_1$ and $f_5 \perp\!\!\!\perp f_2$, then $r_1 = 2/2 = 1$
- n_2: Since $f_4 \perp\!\!\!\perp f_3$ and $f_5 \perp\!\!\!\perp f_2$, then $r_2 = 2/2 = 1$
- n_3: Since $f_4 \perp\!\!\!\perp f_1$, $f_4 \perp\!\!\!\perp f_3$, $f_5 \not\perp\!\!\!\perp f_1$ and $f_5 \not\perp\!\!\!\perp f_3$, then $r_3 = 1/2 = 0.5$

As a result, n_3 is least related neighbor, thus it is removed after the shuffle and replaced by a new neighbor n_5 (Figure 8b).

Paper B [5] gives details on the replication and allocation strategy used by PIQNIC. In short, when a knowledge graph is uploaded to a PIQNIC node n, n fragments the knowledge graph using the predicate-based fragmentation function in Definition 9. Each fragment is sent to a random neighbor which replicates the fragment and passes it on to one of its neighbors. This continues for a certain number of steps called the *replication factor*.

4.3 Query Processing

Due to lack of global knowledge, query processing in PIQNIC follows a *flooding* approach. That is, when a node n issues a query, its triple patterns are forwarded to n's neighbors, which process the triple pattern and forward it to their neighbors, and so on, until the Time-To-Live (TTL) distance has been reached. In the following, we summarize the query processing approach used by PIQNIC given a BGP P by adapting the following steps from [5].

1. Determine the join order of the triple patterns in P using variable counting (triple patterns with less variables are evaluated first).
2. Evaluate P's triple patterns according to the join order by flooding the network with requests using a specified TTL value.
3. Receive partial results from each queried node.
4. Compute the final result by combining the intermediate results of each triple pattern in P.

To determine which fragments contain relevant data to which triple pattern in the query, we formally define the predicate-based identification function (i in Definition 6) as follows:

Definition 12 (Predicate-based identification function – reproduced from [5]) Let $\mathcal{F}_P(\mathcal{G}_N)$ be a the set of fragments in a network, $f \in \mathcal{F}_P(\mathcal{G}_N)$ be a fragment, tp be a triple pattern and p_{tp} the predicate of tp. A predicate-based identification function $\mathcal{F}_{IP}(f, tp)$ returns true iff $\forall t \in f : p_t = p_{tp}$ or p_{tp} is a variable.

To assess the impact of the approach used at step 2 above, we implemented three different strategies that we call `Single`, `Bulk`, and `Full` respectively, which we summarize from [5] as follows:

Single. Inspired by TPF [84], the intermediate result obtained from previously evaluated triple patterns are used to instantiate subsequent triple patterns before flooding the network with requests, one by one.

Bulk. Inspired by brTPF [38], the intermediate bindings from previously evaluated triple patterns are bulked and flooded along with subsequent triple patterns to lower the number of requests.

Full. In contrast to `Single` and `Bulk`, this strategy does not consider the intermediate bindings from previously evaluated triple patterns, but instead processes the original triple patterns in parallel over the network, joining the results afterwards on the issuing node.

4.4 Evaluation and Discussion

The fact that knowledge graphs today have a relatively high degree of unavailability [16, 83] highlights the need for a decentralized solution that ensures the availability of the data even if the data providers fail. To this end, we presented PIQNIC (a P2p clIent for Query processiNg over semantIC data). PIQNIC defines an unstructured P2P architecture in which nodes maintain a local datastore (a set of fragments) and a local view over the network (a set of neighboring nodes). We implemented a prototype of PIQNIC following the approach outlined in this section that allows nodes to join and upload knowledge graphs to a network. Furthermore, the prototype of PIQNIC provides an interface to process queries using three different strategies: `Single`, `Bulk`, and `Full`.

To test the hypothesis that PIQNIC increases the availability of knowledge graphs when nodes fail, and to assess the impact of the query processing strategy on performance, we ran experiments using data and queries from LargeRDFBench [77]. LargeRDFBench consists of 13 interlinked datasets with over a billion triples in total, and 40 queries in 4 different groups: Simple (S), Complex (C), Large Data (L), and Complex and High Number of Sources (CH). The experimental setup and results are available in Paper B, however, in the following we will summarize the results.

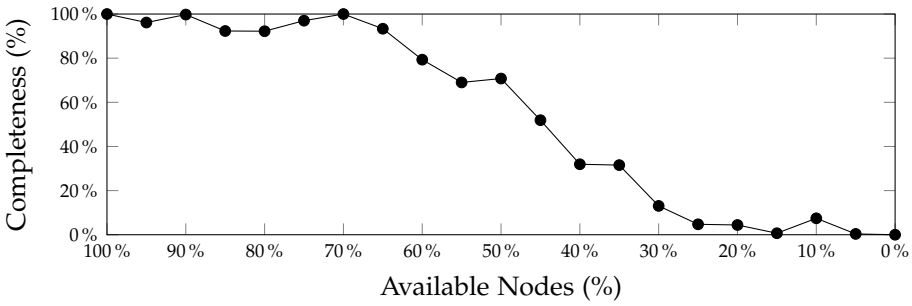


Fig. 9: Completeness (in %) for queries when varying the number of nodes failures (reproduced from [5]).

Figure 9 is reproduced from [5] and shows the robustness of a PIQNIC network when a large number of nodes fail; the x-axis shows the rate of available nodes (40% means that 60% of the nodes were randomly killed) and the y-axis shows the average (over 3 runs) completeness over all queries (i.e., the percentage of computed answers in comparison to the complete set of answers provided by LargeRDFBench) without giving the network any time to recover. As the graph shows, PIQNIC was able to achieve more than 90% completeness even when more than 30% of the nodes were killed, after which the completeness gradually decreased. Furthermore, our experimental results in Paper B [5] showed that the `Bulk` strategy significantly outperformed `Single` and `Full` by incurring less data transfer, both in terms of number of messages and number of bytes sent throughout the network, incurring lower query execution times as a result.

As such, our experimental results show that PIQNIC is able to maintain high availability of the data within the network, even when a large portion of the nodes fail. However, while we assessed the impact of the query processing strategy on performance, the fact that PIQNIC has to flood the network with requests to process queries represents a significant overhead on performance. As such, in the upcoming stages of this thesis, we develop solutions that remove the need for flooding the network by determining, as a pre-processing step, which nodes contain relevant data to a given query.

5 Indexing Decentralized Knowledge Graphs

This section gives an overview of Paper C [6].

5.1 Motivation and Problem Statement

While PIQNIC as presented in Paper B [5] was shown to increase the availability of the data in the network even when a large portion of the nodes fail, the lack of a global overview over the network means that it has to flood the network with requests several times per query execution. And while we already in [5] made some advances towards lowering the impact of flooding the network with requests by sending bulks of intermediate bindings along with each triple pattern request, the flooding mechanism still incurs potentially an exponential number of requests. As such, in Paper C [6], we investigate how flooding can be avoided altogether to allow for more efficient and feasible query processing performance over an unstructured P2P network without a global overview. In particular, we propose two types of indexes called locational indexes and Prefix-Partitioned Bloom Filter (PPBF) indexes respectively, that allow nodes in the network to determine on query time on which nodes relevant data is located, thereby removing the need for flooding the network with requests. In the following, we summarize locational indexes and PPBF indexes presented in Paper C [6].

5.2 Locational Indexes

The indexes presented in Paper C [6] avoid the need for flooding the network with requests at query time by allowing the nodes to estimate where relevant data to each triple pattern in the query is located. In the following, we summarize the formal definition of the baseline *locational* indexes by reproducing the definitions, formulas, and examples with modifications from [6].

Let f be a fragment and $P(f)$ be a function that returns the set of predicates in f. Consider the definition of a node n in Definition 10 as a triple $n = (\Gamma_n, \Delta, N)$ where Γ_n is the set of fragments that n stores [5]. n's locational index $I^i_L(n)$ summarizes the fragments reachable within a distance of i hops from n, and is formally defined in [6] as follows:

Definition 13 (Locational index – adapted from [6])
Let $\mathcal{N}$ be the set of nodes, $\mathcal{P}$ the set of predicates, and Γ the set of fragments, a locational index is a tuple $I^i_L(n)=(\gamma, \eta)$, with $\gamma : \mathcal{P} \rightarrow 2^{\Gamma}$ and $\eta : \Gamma \rightarrow 2^{\mathcal{N}}$. $\gamma(p)$ returns the set of fragments F s.t. $\forall f \in F : p \in P(f)$. $\eta(f)$ returns the set of nodes N such that $f \in \Gamma_{n_i}$ for all $n_i \in N$ such that n_i is within i hops from n.

The following formulas are reproduced from [6]. The locational index at a node n of depth 0, i.e., the index covering only the fragments available locally, is $I_L^0(n) = (\gamma, \eta)$ where:

$$\gamma(p) = \{f \mid f \in \mathcal{F}_n \wedge p \in P(f)\} \tag{9}$$

$$\eta(f) = \{n\}, \forall f \in \mathcal{F}_n \tag{10}$$

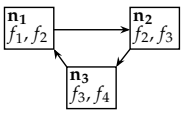
The $\oplus$ operator is reproduced from [6] as follows. $(f \oplus g)(x) = f(x) \cup g(x)$ if f and g are defined at x; $(f \oplus g)(x) = f(x)$ if only f is defined at x; $(f \oplus g)(x) = g(x)$ if only g is defined at x. The locational index of depth i for a node n is defined in [6] as:

$$\gamma = I_L^0(n).\gamma \oplus \bigoplus_{n' \in n.N} I_L^{i-1}(n').\gamma \tag{11}$$

$$\eta = I_L^0(n).\eta \oplus \bigoplus_{n' \in n.N} I_L^{i-1}(n').\eta \tag{12}$$

$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{f_4}$
(a p_1 b)	(a p_2 d)	(c p_3 d)	(b p_2 i)
(f p_1 a)	(b p_2 e)	(g p_3 d)	(g p_2 b)
(c p_1 b)		(e p_3 h)	(b p_1 j)

(a) Fragments and their triples [6]



(b) Connections between nodes [6]

p	$\gamma(p)$
p_1	$\{f_1, f_4\}$
p_2	$\{f_2, f_4\}$
p_3	$\{f_3\}$

f	$\eta(f)$
f_1	$\{n_1\}$
f_2	$\{n_1, n_2\}$
f_3	$\{n_2, n_3\}$
f_4	$\{n_3\}$

(c) $I_L^2(n_1)$ [6]

Fig. 10: Locational index obtained from a set of fragments and connections (reproduced from [6]).

Example 4 (Locational index – adapted from [6])
Given the fragments shown in Figure 10a and the network with the connections and fragment allocations shown in Figure 10b, computing the locational index on node n_1 of depth 2, $I_L^2(n_1)$, is done by applying Equations 11-12. The result is shown in Figure 10c.

Source Selection

Locational indexes let the nodes avoid flooding the network when processing queries by determining which nodes may contain relevant data to each triple pattern. This is done by sending each triple pattern in the query directly to a node that contains a fragment that includes the predicate in the triple pattern. The following example is adapted from [6] and shows how exactly source selection using locational indexes is done.

Example 5 (Source selection – adapted from [6])
Consider the example network on Figure 10b and the locational index on node n_1 in Figure 10c. Processing the triple pattern $tp = (?v_1, p_2, ?v_2)$ on node n_1 incurs processing tp locally on n_1 as well as sending it directly to n_3, since $\gamma(p_2) = \{f_2, f_4\}$, and $n_1 \in \eta(f_2)$ and $n_3 \in \eta(f_4)$.

5.3 Prefix-Partitioned Bloom Filter Indexes

Building on the baseline approach of locational indexes, we now define Prefix-Partitioned Bloom Filter (PPBF) indexes. PPBFs represent the set of IRIs in a fragment as a partitioned bitvector based on Bloom Filters [21]. The idea is that by checking the overlap of PPBFs for fragments that are deemed relevant for triple patterns in a query that join, it is possible to prune non-overlapping fragments (fragments that do not produce join results) from the query execution plan, lowering the number of relevant fragments.

We do not go into detail about Bloom Filters here, but refer the interested reader to [6, 21] for details. Instead, like in [6], we formally define a Bloom Filter $\mathcal{B}$ for a set S with n elements as a tuple $\mathcal{B} = (\hat{b}, H)$ where $\hat{b}$ is a bitvector with m bits and H is a set of k hash functions [21]. In the following, we define PPBFs by reproducing the definitions and examples from [6].

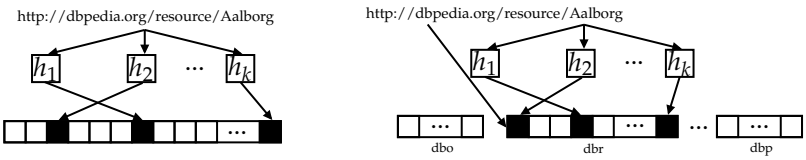
A PPBF is then a Bloom Filter partitioned based on the prefixes (e.g., the IRI `http://dbpedia.org/resource/Aalborg` has the prefix `http://dbpedia.org/resource` [6]) of the IRIs that are inserted, and is formally in [6] defined as follows:

Definition 14 (Prefix-Partitioned Bloom Filter – adapted from [6, 9])
A PPBF $\mathcal{B}^P$ is a 4-tuple $\mathcal{B}^P = (P, \hat{B}, \theta, H)$ where:

- P is a set of prefixes
- $\hat{B}$ is a set of bit vectors such that $\forall \hat{b}_1, \hat{b}_2 \in \hat{B}$, $|\hat{b}_1| = |\hat{b}_2|$
- $\theta : P \rightarrow \hat{B}$ is a prefix-mapping function
- H is a set of hash functions

Furthermore, $\forall p_i \in P$, $\mathcal{B}_i = (\theta(p_i), H)$ is called a partition of $\mathcal{B}^P$ and is the Bloom Filter that encodes the IRIs' names with prefix p_i [6].

Example 6 (PPBF insertion – adapted from [6])
Given the IRI u =`http://dbpedia.org/resource/Aalborg`, inserting u into a regular Bloom Filter is visualized in Figure 11a. Inserting u instead into a PPBF is visualized in Figure 11b. In this case, only the name of the entity is hashed, and the corresponding bits in the partition of its prefix are set to 1.



(a) Regular Bloom Filter [6] **(b)** Prefix-Partitioned Bloom Filter [6]

Fig. 11: Insertion of a IRI into a Bloom Filter and a Prefix-Partitioned Bloom Filter (adapted from [6]).

We say that a IRI u with a prefix p *may* be in a PPBF $\mathcal{B}^P$, denoted $u \sqsubseteq \mathcal{B}^P$, if and only if all the positions determined by the hash functions in $\mathcal{B}^P.H$ applied to u's name are 1 in the bitvector at $\mathcal{B}^P.\theta(p)$ [6]. Given a fragment f, $\mathcal{B}^P(f)$ denotes the PPBF of f. As such, a PPBF index is formalized in [6] similarly to Definition 13 as follows:

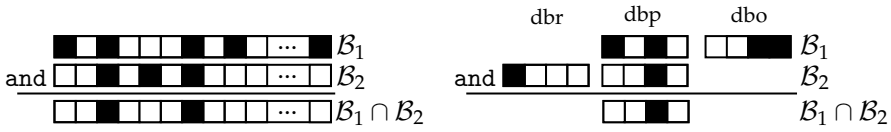
Definition 15 (PPBF Index – adapted from [6])
Let $\mathcal{N}$ be the set of nodes, $\mathcal{U}$ the set of IRIs, and Γ the set of fragments. A PPBF index is a tuple $I_P^i(n) = (\upsilon, \eta)$ with $\upsilon : \mathcal{U} \rightarrow 2^\Gamma$ and $\eta : \Gamma \rightarrow 2^\mathcal{N}$. $\upsilon(u)$ returns the set of fragments f such that $u \sqsubseteq \mathcal{B}^P(f)$. η(f) returns the set of nodes N such that $f \in \mathcal{F}_{n_i}\ \forall n_i \in N$ and n_i is within i hops from n.

Source Selection

Source selection using PPBF indexes involves checking whether or not two fragments for joining triple patterns may produce join results. To do this, we check the overlap of the PPBFs for the fragments by computing the *intersection* of the PPBFs; if the intersection is non-empty, the fragments may produce join results. The intersection of a regular Bloom filter can be found using a bitwise logical `and` operation [53]. PPBF intersection is thus the intersection (logical `and`) of the bitvectors for the prefixes that the two PBBFs have in common, formally defined in [6] as follows:

Definition 16 (PPBF Intersection – adapted from [6])
The intersection of two PPBFs $\mathcal{B}_1^P$ and $\mathcal{B}_2^P$, denoted $\mathcal{B}_1^P \cap \mathcal{B}_2^P$, is $\mathcal{B}_1^P \cap \mathcal{B}_2^P = \langle P_\cap, \hat{B}_\cap, \theta_\cap, H \rangle$, where $P_\cap = \mathcal{B}_1^P.P \cap \mathcal{B}_2^P.P$, $\hat{B}_\cap = \{\mathcal{B}_1^P.\theta(p)$ `and` $\mathcal{B}_2^P.\theta(p) \mid p \in P_\cap\}$, and $\theta_\cap : P_\cap \rightarrow \hat{B}_\cap$.

Example 7 (PPBF Intersection – adapted from [6])
Figure 12a shows an example of intersecting two regular Bloom Filters using the logical `and` operation. The intersection of two PPBFs is visualized in Figure 12b by intersecting the bitvectors for the common prefixes.



(a) Regular Bloom Filters [6] **(b)** PPBFs [6]

Fig. 12: Intersection of Bloom Filters and PPBFs (reproduced from [6]).

For any PPBF index I_P^i there exists a *matching* function that, given a BGP P, returns a *node mapping* M_n that specifies which nodes should be queried for each triple pattern, i.e., $\forall tp \in P$, $M_n(tp)$ is a set of nodes that tp should be processed at. This function is shown in Algorithm 3 which adapts the algorithm from [6].

Algorithm 3 Match BGP To PPBF Index (adapted from [6])

Input: BGP P; Node n; PPBF Index $I_P^i(n) = (v, \eta)$
Output: Node Mapping M_n

1: **function** *matchBGPToPPBFIndex*(P,n,I_P^i)
2: $M_f \leftarrow \{$ (tp, range($I_P^i(n).v$) $\cap \bigcap_{t \in iris(tp)}$ $I_P^i(n).v(t)$) : tp $\in$ P $\}$
3: $M'_f \leftarrow \{$ (tp, $\emptyset$) : $tp \in P$ $\}$
4: **for all** $tp_1, tp_2 \in P$ s.t. $vars(tp_1) \cap vars(tp_2) \neq \emptyset$ **do**
5: $F'_1, F'_2 \leftarrow \emptyset$
6: **for all** (f_1, f_2) s.t. $f_1 \in M_f(tp_1)$ and $f_2 \in M_f(tp_2)$ **do**
7: **if** $\mathcal{B}^P(f_1) \cap \mathcal{B}^P(f_2) \neq \emptyset$ **then**
8: $F'_1 \leftarrow F'_1 \cup \{f_1\}$
9: $F'_2 \leftarrow F'_2 \cup \{f_2\}$
10: **if** $F'_1 \neq \emptyset \wedge F'_2 \neq \emptyset$ **then**
11: $M'_f(tp_1) \leftarrow M'_f(tp_1) \cup \{F'_1\}$
12: $M'_f(tp_2) \leftarrow M'_f(tp_2) \cup \{F'_2\}$
13: **else**
14: $M'_f \leftarrow \{(tp, \emptyset) : tp \in P\}$
15: **break**
16: **return** $\{$ (tp, $\bigcup_{f \in M'_f(tp)}$ takeOne($I_P^i(n).\eta(f)$)) : tp $\in$ P $\}$

In line 2, the algorithm computes the *fragment mapping* M_f that associates triple patterns to the set of fragments in the network that might contain relevant data. For each triple pattern $tp \in P$, $M_f(tp)$ thus returns the set of fragments F such that $\forall u \in iris(tp) : u \sqsubset \mathcal{B}^P(f)$ for all $f \in F$ if $iris(tp)$ is non-empty or $range(I_P^i(n).v)$ otherwise. The for loop in lines 4-15 iterates through each pair of joining triple patterns in P. The PPBFs for each pair of relevant fragments are intersected in line 7; if the intersection is non-empty,

the fragments may produce join results and they are selected in lines 8-9. Once all relevant fragments have been considered, the fragment mapping M'_f is extended with the selected fragments on lines 11-12 or if no fragments were selected, initialized as the empty fragment mapping in line 14. Finally, given the fragment mapping M'_f, line 16 computes the node mapping using the information available in the PPBF index.

5.4 Query Processing

Like in PIQNIC, we process queries triple pattern by triple pattern using bindings from previously evaluated triple patterns similar to brTPF [38] and the `Bulk` query processing approach described in Section 4.3. When processing a query, the first step is *source/fragment selection* using either the locational index or the PPBF index. As such, we now reproduce from [6] operators for retrieving a set of nodes to query given a triple pattern.

Definition 17 (Locational Selection σ^L – adapted from [6])
Let the function $\mathcal{I}(I^i_L(n), p)$ denote the set of nodes that is obtained by using $I^i_L(n)$ to find the relevant nodes to evaluate a triple pattern with predicate p, and $n_1 \in I^i_L(n)$ denote that $n_1 \in \eta(f)$ for some $f \in I^i_L(n).\gamma(p)$. Locational selection for a triple pattern tp on a locational index I^i_L(n) of depth i , denoted $\sigma^L_{tp}(I^i_L(n))$, is the set $\{n_1 \mid n_1 \in I^i_L(n)\}$ if p_{tp} is a variable, or $\{n_1 \mid n_1 \in \mathcal{I}(I^i_L(n), p_{tp})\}$ otherwise.

Definition 18 (PPBF Selection σ^P – adapted from [6])
Let M_n be the node mapping obtained after applying Algorithm 3 to a BGP P. The PPBF selection for a triple pattern $tp \in P$, obtained using the PPBF index $I^i_P(n)$ of depth i, denoted $\sigma^P_{tp,P}(I^i_P(n))$, is the selection of the nodes in $M_n(tp)$.

Evaluating triple patterns relies on evaluating them over the fragments available in the local datastore of the nodes selected using either locational or PPBF selection. We therefore now reproduce from [6] the operator that evaluates a triple pattern over a set of nodes, called *node projection*.

Definition 19 (Node Projection π^N – reproduced from [6])
Given a set of nodes $\mathcal{N}$, node projection on a triple pattern, denoted $\pi^N_{tp}(\mathcal{N})$, is the set of triples obtained by evaluating tp on the local datastore of the nodes in $\mathcal{N}$. Given the function $\mathcal{T}(n, tp)$, that evaluates tp on n's local datastore, node projection is formally defined as: $\pi^N_{tp}(\mathcal{N}) = \bigcup_{n \in \mathcal{N}} \mathcal{T}(n, tp)$

Using the three operators defined above allows the nodes to avoid flooding by applying node projection directly over the nodes specified by either

locational or PPBF selection. As such, building on the approach outlined in Section 4.3, a BGP P is processed using the indexes using the following steps that are adapted from [6]:

1. Determine the join order of P using variable counting.
2. Evaluate each $tp \in P$ using the following steps:
   (a) Apply either locational selection (σ^L) or PPBF selection (σ^P) on n's index to select the nodes $\mathcal{N}$ with relevant fragments for tp.
   (b) Apply node projection (π^N) on $\mathcal{N}$.
3. Compute the final result by combining the intermediate results of each triple pattern in P.

Similar to PIQNIC [5], the iterative process in step 2 is completed by sending bulks of bindings from previously evaluated triple patterns along with the requests [6].

5.5 Evaluation and Discussion

The flooding approach to query processing used by PIQNIC [5] incurs a significant network overhead. In fact, in real cases, the increase in the number of nodes to query for any given triple pattern most likely follows a logistic growth. As such, in Paper C [6], we proposed two new indexing schemes for decentralized architectures such as PIQNIC that avoid the need for flooding the network by determining on which nodes relevant data is located. To test the hypothesis that such indexes lower the communication overhead and in doing so increase query processing performance, We once again ran experiments with the data and queries from LargeRDFBench [77].

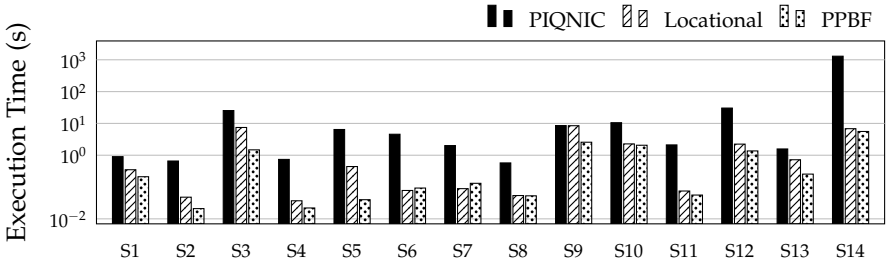
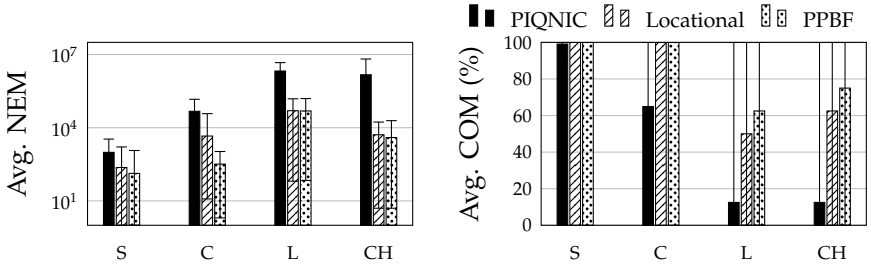


Fig. 13: Execution time for PIQNIC, PIQNIC with locational indexes, and PIQNIC with PPBF indexes over query group S (reproduced from [6]).

The full experimental evaluation can be seen in Paper C [6]; in the remainder of this section, we focus our discussion on the most important experimental results. Figure 13 is reproduced from [6] and shows the execution time for the queries in the S query group from LargeRDFBench. As shown,

locational indexes and PPBF indexes increase performance over all queries; PPBF indexes significantly more so than locational indexes, showing that by pruning fragments that would not otherwise contribute to the overall query result, PPBFs further increase performance. This is also the case for the other query loads.



(a) Average NEM [6] **(b)** Average COM [6]

Fig. 14: Average number of executed messages (NEM) and completeness (COM) for PIQNIC, locational indexes, and PPBFs over query groups (reproduced from [6]).

Figure 14 shows the average number of executed messages (Figure 14a), i.e., the number of requests made between nodes when processing the queries, and completeness (Figure 14b). Clearly, both the locational indexes and the PPBF indexes decrease the network usage significantly for all query loads since they do not have to flood the network with requests. In doing so, they are able to process more of the complex queries without timing out, increasing the completeness overall.

Our experimental results thus show that the indexes reduce the communication overhead when processing queries, increasing query processing performance significantly. While the indexes presented in this section, together with PIQNIC as presented in Section 4 increase the availability of knowledge graphs published on the Web while allowing for relatively efficient access to the data, they still rely on the data providers to keep the data up to date. As such, in Section 6, we build on top of PIQNIC and the decentralized indexes to allow users to collaborate on keeping the data up to date.

6 Collaborative Updates

This section gives an overview of Papers D [7] and E [8].

6.1 Motivation and Problem Statement

The work presented until now has largely focused on increasing the availability and scalability of knowledge graphs on the Web using replication

throughout a P2P network. And while PIQNIC [5] and the decentralized indexes [6] were shown to increase availability while providing relatively efficient access to them, they still rely on the data providers to ensure the data is up to date [7]. As discussed in Section 1.1 and Paper D, this makes difficult to trust the data available in the Web of Data since it can be impossible to (1) trace faulty updates to their origin, (2) ensure consistent query results over long timespans, and (3) update outdated or faulty data. As such, in Paper D, we build on PIQNIC and PPBF indexes and build an architecture that uses blockchain-like chains of updates to knowledge graphs that allow the users to collaborate on keeping the data up to date, called COLCHAIN (COLlaborative knowledge CHAINs) [7]. COLCHAIN splits the entire network into communities of nodes that collaborate on keeping certain data up to date. By structuring the history of updates as a chain and relying on community-wide consensus, COLCHAIN lets users propose updates, trace-back historical updates, and process queries over earlier versions of the data. COLCHAIN is demonstrated in Paper E [8] which provides a fully functioning COLCHAIN client with a graphical UI.

6.2 Preliminaries

While the architecture of COLCHAIN as presented in [7] follows the general fragmentation approach outlined in Section 4.2, we now reproduce a more generalized definition of the distributed indexes from [7].

Definition 20 (Distributed Index – reproduced from [7])
Let n be a node, $\mathcal{N}$ be the set of nodes within a network, $\mathcal{T}$ be the (infinite) set of possible triple patterns, and $\mathcal{F}$ be the (finite) set of fragments that n has access to. A *distributed index* on n is a tuple $I_n = (\nu, \eta)$ with $\nu : \mathcal{T} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^{\mathcal{N}}$. For a triple pattern t, $\nu(t)$ returns the set of fragments in $\mathcal{F}$ that t matches. For a fragment f, $\eta(f)$ returns the nodes on which f is located.

Definition 21 (Node Mapping – reproduced from [7])
For any BGP P and distributed index I, there exists a function $match(P, I)$ that returns a *node mapping* $M : T \mapsto 2^{\mathcal{N}}$ where T is the set of triple patterns in P, such that $M(t)$ returns the nodes that have fragments matching t.

Nodes share partial indexes for each fragment, called index *slices* that are the part of the distributed index that describes the particular fragment. Index slices are formally defined in [7] as follows:

Definition 22 (Index Slice – reproduced from [7])
Let f be a fragment. A *slice* of f, s_f, is a tuple $s_f = (\nu', \eta')$ where $\nu'(t)$ returns f if there exists a triple in f that matches t, and $\eta'(f)$ returns the set of all nodes that contain f in their local datastore.

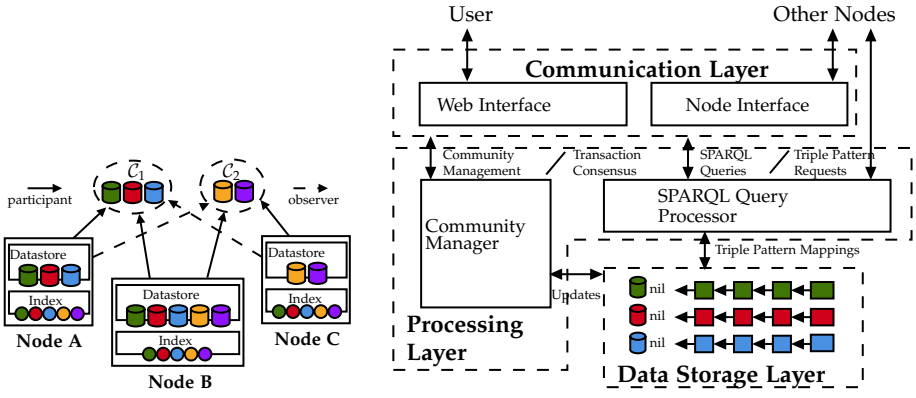
The function $s(f)$ returns the slice of f. Nodes download the slices S of the fragments they index (the indexing strategy is outlined in Section 6.3) and combine them using the following formula [7]:

$$I(S) = \left(\bigoplus_{s \in S} s.\nu', \bigoplus_{s \in S} s.\eta' \right) \tag{13}$$

While the definition of the distributed indexes in this section theoretically allows for any type of indexes, our implementation of ColChain uses PPBF indexes described in Section 5.3, and we adapt our definitions from [7] accordingly. Thus, an index slice of a fragment is the PPBF of that fragment. Indeed, our implementation of ColChain uses PPBF indexes.

6.3 ColChain: Collaborative Linked Data Networks

ColChain introduces the notion of *communities*; the entire P2P network is divided into smaller communities of nodes that collaborate on keeping a certain set of knowledge graphs up to date. This is to allow for secure consensual updates without the need for a consensus across the entire network. Take, for instance the example ColChain network shown in Figure 15a that consists of two such communities and three nodes. The nodes can either *participate* in or *observe* a community. Participants store the knowledge graphs in their local datastore and participate in keeping the data up to date, while observers just include the knowledge graphs in their index. In Figure 15a, node A participates in community $\mathcal{C}_1$ and thus replicates the fragments from that community, while it observes community $\mathcal{C}_2$ and thus indexes the data from that community.



(a) Example ColChain network **(b)** Architecture of a ColChain node

Fig. 15: An example ColChain network and the architecture of a node (reproduced from [7, 8]).

To facilitate consensual updates, ColChain represents the history of updates to a fragment as a chain of so-called *transactions*, i.e., sets of bundled op-

erations such as additions or deletions of triples, similar to blockchains [69]. As such, like with blockchains, ColChain relies on the consensus of nodes within the community to accept transactions. This allows any node to propose an update to a fragment while ensuring malicious updates have a low likelyhood of being pushed to the chain. Furthermore, by representing the history of updates to a fragment as an update *chain*, ColChain lets users roll-back updates and process queries over the version of a fragment available at a certain timestamp, as well as tracing back faulty updates. While the consensus protocol allows any node to propose updates, it also allows the *owner* to enforce updates using the RSA [75] digital signature scheme.

Figure 15b shows the architecture of node A from Figure 15a. It combines the conventional data storage layer from regular P2P systems (Section 3) with the blockchain layer in its local datastore. The query processor is able to process queries over any historical version of the fragments by rolling the fragments back on query time using the update chain. The node interface defines an interface that nodes can use to communicate with one another, e.g., when obtaining consensus on an update.

Formal Definition

A ColChain network N is a set of nodes $N = \{n_1, \ldots, n_n\}$ that maintain a local datastore with fragments from each community they participate in. In the following, we formally define a node and the state of a node by reproducing the definitions from [7].

Definition 23 (Node – reproduced from [7])
A node n is a triple $n = (K, a_n, u_n)$ where

- $K = (\kappa_n, \rho_n)$ is a key-pair such that κ_n is n's private key and ρ_n is n's public key
- a_n is n's address
- u_n is a valid IRI and n's unique identifier

The key-pair K describe the private and public keys of a node and are used in the consensus protocol to check for fragment ownership. The state of a node is defined as follows:

Definition 24 (Node State – reproduced from [7])
Let n be a node. n's *state* is $\mathcal{S}_n = (\Sigma, S, I_n, \mathcal{M})$ where

- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ and $\forall \sigma_i \in \Sigma : \sigma_i = (\mathcal{X}_i, f_i)$ such that
  - $\mathcal{X}_i$ is a secure hash chain
  - f_i is a fragment
  - All updates represented in $\mathcal{X}_i$ have been applied to f_i
  - All f_j, f_k such that $1 \leq j, k \leq m$ and $j \neq k$ refer to different (unique) fragments in n's local datastore

  - σ_i is said to be an *entry* in n's local datastore
- S is a set of index slices and $\forall \sigma_i \in \Sigma : s(f_i) \in S$
- I_n is n's distributed index, $I_n = I(S)$, consisting of index slices from local and remote fragments
- $\mathcal{M}$ is a set of *metadata* triples

The metadata triples describe the node's local view over the network and include the metadata of all communities the node participates in or observes (Definition 26) [7]. For instance, they include the triples describing other nodes within the community as well as fragment ownership. Notice that in the case that the node observes a community, the index slices S contain slices of the fragments within the community even though they are not in the local datastore. A community is defined formally as follows:

Definition 25 (Community – reproduced from [7])
A community $\mathcal{C}$ contains two sets of nodes called participants and observers (defined in Definitions 27 and 28) and a set of fragments, each owned by a node, i.e., $\mathcal{C} = (N, F_\mathcal{C}, v, u_\mathcal{C})$ where

- $N = (P_\mathcal{C}, O_\mathcal{C})$ such that $P_\mathcal{C}$ and $O_\mathcal{C}$ are the sets of participants and observers, respectively
- $F_\mathcal{C}$ is a set of fragments
- v is an *ownership-mapping* function such that $\forall f \in F_\mathcal{C} : v(f) = \rho_f$ where $\exists n \in P_\mathcal{C} \bigcup O_\mathcal{C} : \rho_f = \rho_n$
- $u_\mathcal{C}$ is a valid IRI and $\mathcal{C}$'s unique identifier

The ownership-mapping function v maps fragments to their owners and is used in the consensus protocol [7]. Given a fragment f, let u_f denote the unique identifier for f such that u_f is a valid IRI. The state of a community is formally defined in [7] as follows:

Definition 26 (Community State – reproduced from [7])
Let $\mathcal{C}$ be a community. $\mathcal{C}$'s *state* is $\mathcal{S}_\mathcal{C} = (\Phi, \mathcal{M})$ where

- $\Phi = \{\phi_1, \phi_2, \ldots, \phi_m\}$ and $\forall \phi_i \in \Phi : \phi_i = (\mathcal{X}_i, f_i)$
  - $\mathcal{X}_i$ is a secure hash chain
  - f_i is a fragment
- $\mathcal{M}$ is a set of *metadata* triples such that
  - $\forall n \in P_\mathcal{C} : (u_\mathcal{C}, \texttt{cc} : \texttt{participant}, u_n) \in \mathcal{M}$
  - $\forall n \in O_\mathcal{C} : (u_\mathcal{C}, \texttt{cc} : \texttt{observer}, u_n) \in \mathcal{M}$
  - $\forall f \in F_\mathcal{C} : (u_\mathcal{C}, \texttt{cc} : \texttt{fragment}, u_f) \in \mathcal{M}$ and $(u_f, \texttt{cc} : \texttt{author}, \mathcal{C}.v(f)) \in \mathcal{M}$

Since nodes can both participate in and observe a community, we now define community participation and observation by reproducing the following definitions from [7].

Definition 27 (Participant – reproduced from [7])
Given a community $\mathcal{C}$ and a node n, n is said to be a *participant* in $\mathcal{C}$ iff for all $\phi = (\mathcal{X}, f) \in \mathcal{S}_{\mathcal{C}}.\Phi$ it is the case that $(\mathcal{X}, f) \in \mathcal{S}_n.\Sigma$, $s(f) \in \mathcal{S}_n.S$, and $\forall t \in \mathcal{S}_{\mathcal{C}}.\mathcal{M} : t \in \mathcal{S}_n.\mathcal{M}$.

As such, a participant is a node that stores the fragments from the community within its local datastore. Nodes can participate in multiple communities at once. Furthermore, participants within a community collaborate on keeping the data up to date. An observer is defined formally in [7] as follows:

Definition 28 (Observer – reproduced from [7])
Given a community $\mathcal{C}$ and a node n, n is said to be an *observer* in $\mathcal{C}$ iff for all $\phi = (\mathcal{X}, f) \in \mathcal{S}_{\mathcal{C}}.\Phi$ it is the case that $s(f) \in \mathcal{S}_n.S$, and $\forall t \in \mathcal{S}_{\mathcal{C}}.\mathcal{M} : t \in \mathcal{S}_n.\mathcal{M}$.

By observing a community, nodes thus gain access to the data without having to replicate all the fragments, increasing query completeness. Observers thus access the data by requesting the relevant data from a participant on query time.

Consensual Updates

ColChain represents updates as *transactions* that consist of a set of *operations*. An operation is either the addition or deletion of a triple. We now reproduce the formal definition of transactions and operations from [7]. An operation is in [7] defined as follows:

Definition 29 (Operation – reproduced from [7]))
An operation o over a fragment f is a tuple $o = (\alpha, t)$, where α describes whether the operation is an insertion ($\alpha = +$) or deletion ($\alpha = -$) of a triple, and t is the triple that is to be inserted or deleted.

Given an operation $o = (\alpha, t)$ and a fragment f, $o(f)$ denotes the result of applying o to f, i.e., the fragment that does not contain t if $\alpha = -$ or the fragment that does contain t if $\alpha = +$. Since operations typically occur in sets, we reproduce the formal definition of transactions from [7] as a set of operations including information about which fragment it is applied to as well as provenance, as follows:

Definition 30 (Transaction – reproduced from [7])
A transaction γ is a triple $\gamma = (O, f_i, \lambda)$ where $O = \{o_1, o_2, \ldots, o_n\}$ is a set of operations, f_i is a fragment, and $\lambda = (u_n, \alpha_\gamma, \iota)$ is provenance information, where u_n is the unique identifier of the node proposing the transaction n, α_γ is a signature obtained using n's private key, and ι is a timestamp.

Given a transaction γ and fragment f, applying all operations in γ to f is denoted $[[f]]_\gamma$.

Table 2: Example of applying a transaction to a fragment (adapted from [5])

(a) Fragment f

Fragment f		
$(\mathbf{a}, \mathbf{p_1}, \mathbf{c})$	(a, p_2, d)	(b, p_2, d)
(b, p_3, e)	(c, p_1, d)	(c, p_3, a)
(f, p_4, d)	(d, p_4, f)	(e, p_5, g)

(b) The result after applying γ to f, $[[f]]_\gamma$

Fragment $[[f]]_\gamma$		
$(\mathbf{a}, \mathbf{p_1}, \mathbf{b})$	(a, p_2, d)	(b, p_2, d)
(b, p_3, e)	(c, p_1, d)	(c, p_3, a)
(f, p_4, d)	(d, p_4, f)	(e, p_5, g)

Example 8 (Transaction – adapted from [7])
Consider fragment f in Table 2a and the following transaction:

$$\gamma = (\{(+, (a, p_1, b)), (-, (a, p_1, c))\}, f, \lambda)$$

The result of applying γ to f, $[[f]]_\gamma$, is the fragment that exchanges the triple (a, p_1, c) with (a, p_1, b), and can be seen in Table 2b.

Next, we reproduce the state transition functions from [7] that define how updates are applied to a node when a consensus has been reached, both for the fragments as well as the index slices. The fragment transition function is defined as follows [7]:

Definition 31 (Fragment Transition Function – reproduced from [7])
Given a transaction γ, a state $\mathcal{S}_n$ of a node n, and a secure hash function H, the *fragment transition function* is for all $\sigma_i = (\mathcal{X}_i, f_i) \in \mathcal{S}_n.\Sigma$ defined as follows.

$$\tau_\Sigma(\sigma_i, \gamma) = (\tau_\mathcal{X}(\mathcal{X}_i, \gamma), \tau_f(f_i, \gamma)) \tag{14}$$

Where

$$\begin{aligned} \tau_\mathcal{X}(\mathcal{X}_i, \gamma) = \mathcal{X}_i \cup \{x_{i+1}\} \text{ s.t.} \\ x_{i+1} = (h, \gamma) \\ h = (H(\gamma), y) \\ y = \{H(x_j) \mid j < i\} \end{aligned} \tag{15}$$

And

$$\tau_f(f_i, \gamma) = [[f_i]]_\gamma \tag{16}$$

h in Equation 15 is a *header* that contains the hash value of the transaction and a link to the header of the previous transaction, forming a chain [7]. The slice transition function is defined in [7] as follows:

Definition 32 (Slice Transition Function – reproduced from [7])
Given a transaction $\gamma = (O, f_i, \lambda)$ and a state $\mathcal{S}_n$ of a node n such that $s_{f_i} \in \mathcal{S}_n.S$, the *slice transition function* is defined as follows.

$$\tau_S(s_{f_i}, \gamma) = s([[f_i]]_\gamma) \tag{17}$$

When a transaction is applied, the participants first apply the fragment transition function and then compute the new index slice by applying the slice transition function.

The consensus protocol followed by ColChain relies on either a majority vote across the participants in the community, or checking if it is the owner that proposed the transaction, called Proof-of-Ownership (PoO). Let $verify(\alpha_\gamma, \rho_f)$ be a function that returns `true` if and only if the signature α_γ matches the owner's public key ρ_f, and $validate(\gamma)$ be a function that returns `true` if and only if the user accepts γ. The *acceptance* protocol then follows the following steps adapted from [7]:

1. Let $\gamma = (O, f, \alpha_\gamma)$, $\mathcal{M} = \mathcal{S}_n.\mathcal{M}$, and $\sigma_i = (\mathcal{X}_i, f_i) \in \mathcal{S}_n.\Sigma$ be the unique state entry such that $u_{f_i} = u_f$. Find $\langle u_f, \texttt{cc} : \texttt{author}, \rho_f \rangle \in \mathcal{M}$.
2. If $verify(\alpha_\gamma, \rho_f) = \texttt{true}$, accept γ on σ_i.
3. If $validate(\gamma) = \texttt{true}$, accept γ on σ_i.
4. If $verify(\alpha_\gamma, \rho_f) = \texttt{false}$ and $validate(\gamma) = \texttt{false}$, reject γ on σ_i.

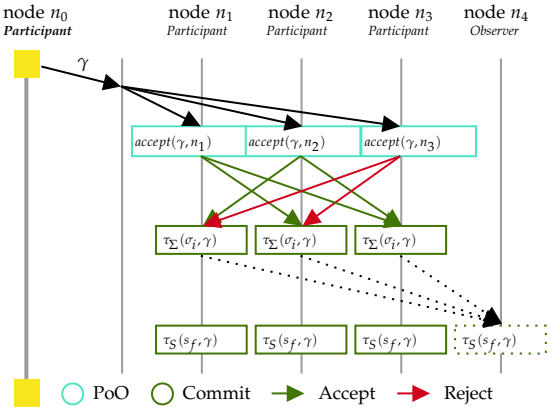


Fig. 16: Flowchart of proposing, validating and applying (reproduced from [7]).

Example 9 (Consensus Protocol – adapted from [7])
Consider a community $\mathcal{C}$ that has four participants, $\{n_0, n_1, n_2, n_3\}$ and one observer $\{n_4\}$, and a transaction γ proposed by n_0. Figure 16 shows the flowchart of what happens when n_0 proposes γ; first, each participant follows the acceptance protocol outlined above and determines whether or not to accept the transaction. In this case, n_1 and n_2 vote to accept the transaction while n_3 votes to reject it. Since this is a majority, all participants call the fragment transition function locally. Then, they call the slice transition function to update the index slice and propagate the new slice to the observers (n_4).

6.4 Query Processing

Similarly to PIQNIC [5] with the PPBF indexes [6], and in line with related work [38], COLCHAIN attaches bindings from previously evaluated triple patterns to subsequent requests in bulks. Recall in Definition 21 the function $match(P, I)$ that returns a node mapping M that maps triple patterns to the set of nodes that contain relevant fragments to that triple pattern. With PPBF indexes, this function corresponds to the function shown in Algorithm 3.

In line with previous work [38, 84], each node n contains a *selector function* that defines how triple pattern requests are processed over an entry in n's local datastore [7]. To accommodate processing queries over earlier versions of a fragment, we extend the selector function defined in [38] with the notion of a timestamp ι. Let $\sigma_i = (\mathcal{X}_i, f_i)$ be an entry in n's local datastore. $\iota(\sigma_i)$ thus denotes the fragment obtained by reversing the operations in each transaction in $\gamma_j \in \mathcal{X}_i$ such that $\gamma_j.\lambda.\iota \leq \iota$, resulting in the fragment what was available at ι. The selector function is then defined in [7] as follows:

Definition 33 (Selector Function – reproduced from [7])
Given a node n, a triple pattern tp, a finite set of distinct solution mappings Ω, and a timestamp ι, the fragment-based bindings-restricted triple pattern selector for tp, Ω, and ι, denoted $s_{(tp,\Omega,\iota)}$, is for every entry σ in n's local datastore defined in [7] as follows:

$$s_{(tp,\Omega,\iota)}(\sigma) = \begin{cases} \{t \in \iota(\sigma) \mid t \in \iota(\sigma) \wedge \exists \mu : t = \mu[tp] & \text{if } \Omega = \emptyset \\ \{t \in \iota(\sigma) \mid t \in \iota(\sigma) \wedge \exists \mu : t = \mu[tp] \wedge & \\ \qquad \exists \mu' \in \Omega : \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

Given a node n, a triple pattern tp, a set of solution mappings Ω, and a timestamp ι, let $n^c(tp, \Omega, \iota)$ be a function that returns the cardinality estimation of the result of invoking $s_{(tp,\Omega,\iota)}$ on n and $n^p(tp, \Omega, \iota)$ a function that returns the result of invoking $s_{(tp,\Omega,\iota)}$ on n [7].

Algorithm 4 shows a recursive algorithm that processes a BGP P over the fragments covered by the index of a node n. In line 2, the algorithm computes the node mapping for P (Definition 21). Then, the for loop in lines 3-5 computes the estimated cardinality of each triple pattern in P as the sum of the cardinality estimation over each node specified by the node mapping; the triple pattern with the lowest estimated cardinality is selected in line 6 and sent to the nodes specified by the node mapping in line 7. The resulting bindings are joined with the intermediate bindings from previously evaluated triple patterns in line 8. Last, the algorithm makes a recursive call in line 11 if there are any remaining triple patterns in P.

Algorithm 4 Evaluate a BGP on a ColChain node – reproduced from [7]

Input: A BGP $P = \{tp_1, \ldots, tp_n\}$; a node n; a timestamp ι; a set of solution mappings Ω; a node mapping M
Output: A set of solution mappings

function *evaluateBGP*(P,n,ι,$\Omega = \emptyset$,$M = \emptyset$)
  if $M = \emptyset$ **then** $M \leftarrow match(P, \mathcal{S}_n.I_n)$;
  for all $tp_i \in P$ **do**
    $cnt_i \leftarrow \Sigma_{n_1 \in M(tp_i)} n_1^c(tp_i, \Omega, \iota)$;
    if $cnt_i = 0$ **then return** $\emptyset$;
  $tp_\epsilon \leftarrow tp_k$ where $tp_k \in P$ and $cnt_k \leq cnt_j \forall tp_j \in P$;
  $\phi^\epsilon \leftarrow \bigcup_{n_1 \in M(tp_\epsilon)} n_1^p(tp_\epsilon, \Omega, \iota)$;
  $\Omega^\epsilon \leftarrow \Omega \bowtie \{\mu \mid dom(\mu) = vars(tp_\epsilon)$ and $\mu[tp_\epsilon] \in \phi^\epsilon\}$;
  $P' \leftarrow P \setminus \{tp_\epsilon\}$;
  if $P' = \emptyset$ **then return** Ω^ϵ;
  return $evaluateBGP(P', n, \iota, \Omega^\epsilon, M)$;

6.5 Demonstration

The approach outlined in this section was demonstrated in Paper E [8] which introduces a fully functioning ColChain client along with a graphical UI on top. In the GUI, users are able to process queries, manage communities, as well as propose and vote on updates. Figure 17 shows a screenshot of the GUI of ColChain in a situation where the user is proposing an update that changes the current (incumbent) US President from `dbr:Donald_Trump` to `dbr:Joe_Biden` following the inauguration of President Biden. The following description is adapted from [8].

To update the current US President, the user first searches for the IRI `dbr:President_of_the_United_States` on Figure 17a and thus finds the triple where `dbr:Donald_Trump` is the object (Figure 17b), which they then remove. In the next step, the user then adds the same triple with `dbr:Joe_Biden` as the object on Figure 17c. Figure 17d then shows the proposed changes. Upon saving the update (Figure 17e), the node forwards the transaction to the other participants, all of which are notified as shown on Figure 17f.

During the conference, attendees were able to explore ColChain in two separate scenarios. First, we ran a network with the data from LargeRDFBench [77], and second we ran a network with a subset of the DBpedia [17] data with update chains dating back to version 2015-04. To ease the interaction from attendees, we prepared several interesting SPARQL queries and also allowed the attendees to write their own queries, as well as navigate the interface and propose updates to the fragments.

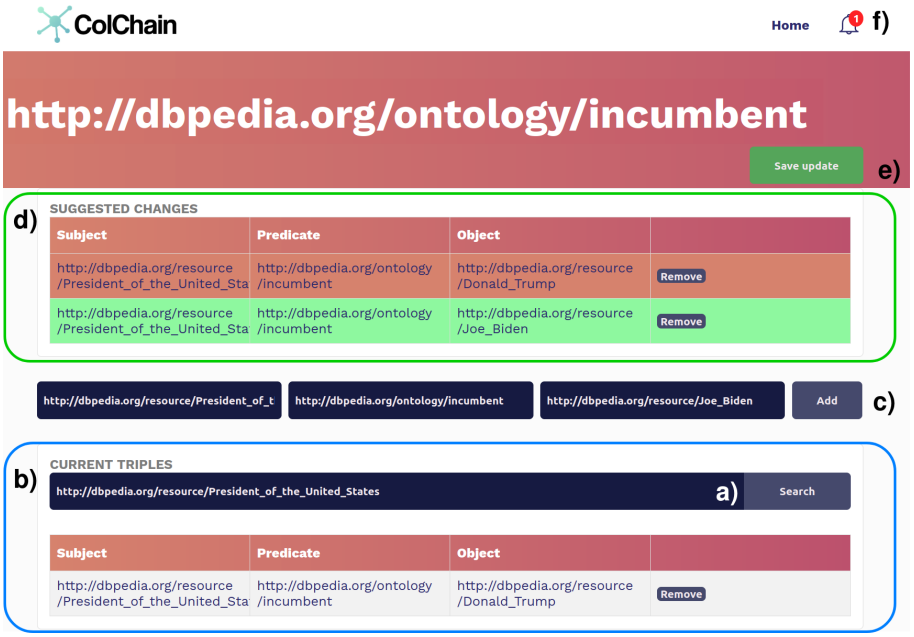


Fig. 17: ColChain's graphical interface when proposing an update to a fragment (reproduced from [8]).

6.6 Evaluation and Discussion

While the approach outlined in Sections 4 and 5 increase the availability of knowledge graphs on the Web, they still rely on the data providers to keep the data up to date. To address the updatability issue, we introduced ColChain (COLlaborative knowledge CHAINs) in Paper D [7], which divides the entire network into communities of nodes that collaborate on keeping certain data up to date by relying on consensus, and demonstrated a fully functioning ColChain client along with a graphical UI in Paper E [8]. By structuring the history of updates over the fragments as a chain, ColChain further allows the nodes within a community to roll back updates and process queries over an earlier version.

In doing so, ColChain solves the second major issue the Web of Data faces by increasing the updatability of the data. To test the hypotheses that ColChain does so without bringing a major cost on query processing performance, and that it is able to process queries over earlier versions of the data, we tested our approach against Piqnic and Piqnic with the PPBF indexes (called Piqnic_{PPBF}) using the data and queries from LargeRDFBench [77]. The full experimental results can be seen in [7], however in this discussion, we reproduce some of the most important experimental results.

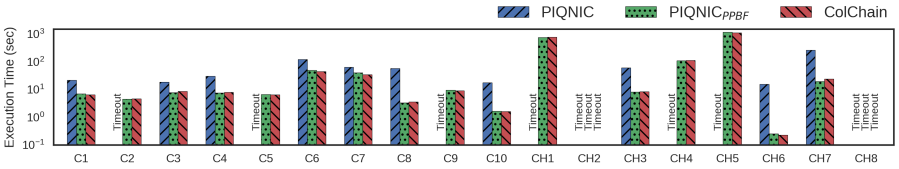
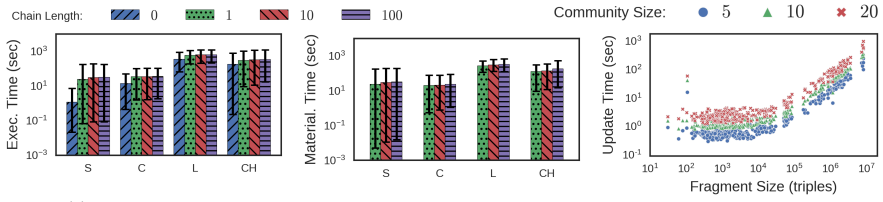


Fig. 18: Execution time for queries in the C and CH categories for PIQNIC, PIQNIC$_{PPBF}$, and COLCHAIN (reproduced from [7]).

Figure 18 shows the execution time for the queries over the C and CH query loads. As evident, COLCHAIN has similar performance as PIQNIC$_{PPBF}$ across the board; this is also the case for the remaining query loads, and shows that the additional functionality that COLCHAIN provides does not negatively affect performance. This is the case since network traffic is not negatively affected; COLCHAIN incurs a similar network cost as PIQNIC$_{PPBF}$ [7]. The experimental results presented in [7] further show that the size of the communities has an impact as well; since larger communities means a higher replication factor (i.e., fragments are replicated across more nodes), there is a higher chance that relevant data is available locally on the issuing node, lowering the network overhead and increasing performance.



(a) Execution Time

(b) Materialization Time

(c) Update Overhead Time

Fig. 19: Execution and materialization time excluding queries that timed out, and update overhead time over different community sizes (reproduced from [7]).

Figure 19 shows the execution (Figure 19a) and fragment materialization (Figure 19b) times when processing queries over earlier versions for update chains of various lengths. Evidently, the length of the chain has a negligible effect on performance. In fact, this effect is sub-linear, since materializing each version in-between the current and target versions is unnecessary; it is sufficient to compute the changes between the two versions and materialize the older fragment only once. Since COLCHAIN uses HDT [29] to store fragments, which does not support dynamic updates, it has to decompress the triples in the current version of a fragment and compress the triples in the target version when materializing the older dataset versions. As such, the majority of the materialization time is spent on decompression and compres-

sion of the triples in the target fragment (94% of the time in our experiments). Figure 19c shows the time it takes to apply updates to each fragment, i.e., the time it takes to complete the consensus protocol outlined in Section 6.3, for different community sizes. The majority of the update time is spent on materialization and the size of the communities has limited impact on the overhead time.

Overall, our experiments show that the main limitation wrt. the performance of processing queries over earlier versions as well as applying updates to fragments, is the usage of HDT as the backend; in the future, it could be interesting to use an RDF compression technique that allows for dynamically updating the data to assess the impact on performance. As such, ColChain is able to let users collaborate on keeping the data up to date without incurring a significant cost on query processing performance, and is able to provide query processing over earlier fragment versions with good scalability. ColChain thus proposes solutions to all three of the major issues discussed in Section 1.1; availability, updatability, and scalability. However, even still ColChain does not improve upon the query processing efficiency of Piqnic [5], and therefore suffers from the same issues; while the query processing efficiency has partly been addressed by the indexes presented in Paper C [6], they still have a relatively high network overhead. As such, in Section 7, we present a novel approach to lowering the communication overhead and, in doing so, optimizing queries over Piqnic and ColChain.

7 Optimizing Decentralized SPARQL Queries

This section gives an overview of Paper F [9].

7.1 Motivation and Problem Statement

While ColChain [7] extends Piqnic [5] to also allow users to collaborate on keeping the data up to date, and thereby solves both the availability and updatability problems, it does not improve query processing efficiency or scalability compared to Piqnic. As such, it still suffers from some of the same pitfalls that Piqnic does; namely that processing queries in such a setup can be quite challenging and puts a strain on the entire network, since a large number of nodes have to be queried for relevant data to a single query. While the indexes presented in Paper C [6] go some way in decreasing the network overhead compared to the flooding approach used by Piqnic, there is still a relatively large network overhead that could be decreased further. As such, in Paper F [9], we present Lothbrok, an approach to optimizing SPARQL queries over decentralized knowledge graphs that fragments the data based on characteristic sets [70] and adapts the indexes presented in [6] to the new

fragmentation approach. To optimize queries over such fragments, LOTHBROK decomposes the query into star-shaped subqueries similar to WISEKG [18] and uses a query optimization strategy based on the locality of data, fragment compatibility, i.e., whether or not two fragments produce join results, and cardinality estimations enabled by the indexes.

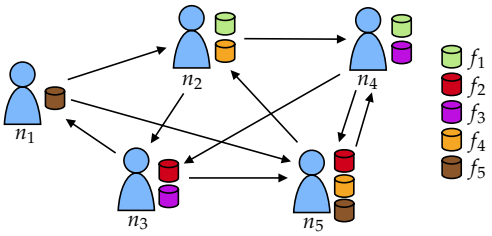


Fig. 20: Example P2P network (reproduced from [9]).

7.2 The Lothbrok approach

To increase performance in P2P systems like PIQNIC [5] and COLCHAIN [7], LOTHBROK makes three contributions that work together to decrease the communication overhead while spreading the query processing effort more evenly across multiple nodes. In the following, we describe how LOTHBROK structures the data by reproducing the definitions and examples describing data fragmentation and indexing from [9].

First, LOTHBROK proposes to fragment the data based on characteristic sets as defined in Definition 5, such that entire star-shaped subqueries can be answered by a single fragment. This fragmentation approach is used, since processing multiple star patterns concurrently can decrease the network overhead [4] while star-shaped subqueries are relatively efficient for the nodes to process [72].

> **Example 10 (Characteristic sets – adapted from [9])**
> Consider, as an example, the P2P network in Figure 20 and the query Q shown in Figure 21a. Table 21b shows the characteristic sets of each fragment in the network; in this example, $P_3 \in \mathcal{S}(Q)$ (Definition 3) can be processed entirely over f_5.

Second, LOTHBROK defines a novel indexing scheme that encodes structural information useful for cardinality estimation and locality awareness and that takes advantage of the fragmentation based on characteristic sets, called Semantically Partitioned Bloom Filter (SPBF) indexes that build on PPBF indexes (Section 5).

```
select * where {
  ?person dbo: nationality ?country . # tp1 (P1)
  ?person dbo:author ? publication . # tp2 (P1)
  ?country dbo: capital ? capital . # tp3 (P2)
  ?country dbo:currency ?currency . # tp4 (P2)
  ? publication dbo: publisher ? publisher . #tp5 (P3)
  ? publication dbo:language ?language . #tp6 (P3)
}
```

(a) Query Q

Fragment	CS
f_1	{dbo:nationality,dbo:author,dbo:deathDate}
f_2	{dbo:nationality,dbo:author}
f_3	{dbo:capital,dbo:currency,dbo:population}
f_4	{dbo:capital,dbo:currency}
f_5	{dbo:publisher,dbo:language}

(b) CSs of each fragment in the running example

Fig. 21: (a) Example SPARQL query Q and (b) corresponding characteristic sets in the example network (reproduced from [9]).

Third, Lothbrok proposes a query optimization strategy that takes advantage of the characteristic set based fragments and SPBF indexes to compute a query execution plan in consideration of fragment compatibility, cardinality estimations, and locality information.

Data Fragmentation

Given the formal definition of characteristic sets from Definition 5, we define the characteristic set fragmentation function as a fragmentation function that, given a knowledge graph $\mathcal{G}$, creates one fragment for each unique characteristic set in $\mathcal{G}$. Formally, this is defined as follows:

Definition 34 (CS Fragmentation Function – reproduced from [9])
Let $\mathcal{G}$ be a knowledge graph, then the *characteristic set fragmentation function* of $\mathcal{G}$, $\mathcal{F}_C(\mathcal{G})$, is defined using the notation introduced in Definition 5, as:

$$\mathcal{F}_C(\mathcal{G}) = \{\{(s,p,o) \mid (s,p,o) \in \mathcal{G} \wedge C_{\mathcal{G}}(s) = C_i\} \mid C_i \in C(\mathcal{G})\} \tag{18}$$

For instance, fragment f_4 in Table 21b contains all triples of the subjects that are described by the characteristic set `{dbo:capital,dbo:currency}`. However, to avoid a huge number of fragments for relatively unstructured knowledge graphs, we *normalize* characteristic sets as follows (the following description is adapted from [9]): First, we merge a fragment f_1 with characteristic set C_1 into a fragment f_2 with characteristic set C_2 if $C_1 \subseteq C_2$ by adding the triples from f_1 to f_2. If there are multiple candidates for f_2, we select the fragment with the smallest characteristic set (the smallest number of predicates). Second, we split fragments f with infrequent characteristic sets (below a certain number of subjects) into two separate fragments f' and f'' that can individually be merged into fragments with more frequent characteristic sets by following the first step.

SPBF Indexes

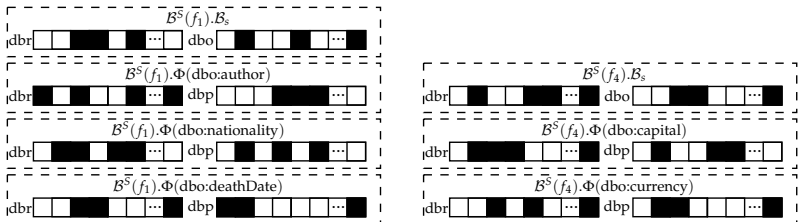
To accommodate the characteristic set fragmentation, and to efficiently estimate whether or not two fragments produce join results for a particular

query, LOTHBROK uses SPBF indexes that build on the baseline approach defined by PPBF indexes (Section 5). In particular, SPBFs encode the set of subjects in a fragment as a single prefix-partitioned bitvector (PPBF), while there is one prefix-partitioned bitvector for each predicate value that encodes the objects associated with that predicate [9]. Formally, this is defined as follows [9]:

Definition 35 (Semantically Partitioned Bloom Filter – reproduced from [9]) An SPBF $\mathcal{B}^S$ is a 5-tuple $\mathcal{B}^S = (P, \mathcal{B}_s, B_o, \Phi, H)$ where:

- P is a set of distinct predicate values
- $\mathcal{B}_s$ is the prefix-partitioned bitvector that summarizes the subjects
- B_o is the set of prefix-partitioned bitvectors that summarize the objects
- $\forall \mathcal{B}_i \in \{\mathcal{B}_s\} \cup B_o$, $\mathcal{B}_i = (P_i, \hat{B}_i, \theta_i)$ where:
  - P_i is a set of prefixes
  - $\hat{B}_i$ is a set of bitvectors such that $\forall \hat{b}_1, \hat{b}_2 \in \hat{B}_i : |\hat{b}_1| = |\hat{b}_2|$
  - $\theta_i : P_i \rightarrow \hat{B}_i$ is a prefix-mapping function
- $\Phi : P \rightarrow B_o$ is a predicate-mapping function s.t. $\forall p \in P : \Phi(p) \in B_o$
- H is a set of hash functions

Example 11 (SPBFs – adapted from [9])
Consider the network shown in Figure 20. Figure 22 shows the SPBFs of fragments f_1 (Figure 22a) and f_4 (Figure 22b); as shown, the SPBF of each fragment contains a prefix-partitioned bitvector describing the subject values in the fragments, and one prefix-partitioned bitvector for each predicate in the fragment describing the objects associated with that predicate.



(a) SPBF $\mathcal{B}^S(f_1)$ **(b)** SPBF $\mathcal{B}^S(f_4)$

Fig. 22: SPBFs of (a) f_1, $\mathcal{B}^S(f_1)$, and (b) f_4, $\mathcal{B}^S(f_4)$, in the running example (reproduced from [9]).

Similarly to PPBFs, we say in [9] that an IRI u at position $\rho \in \{s, p, o\}$ *may* be in an SPBF $\mathcal{B}^S$, denoted $u \sqsubset^{\rho} \mathcal{B}^S$, if $u \sqsubset \mathcal{B}^S.\mathcal{B}_s$ if $\rho = s$, $u \in \mathcal{B}^S.P$ if $\rho = p$, or $\exists p \in \mathcal{B}^S.P$ where $u \sqsubset \mathcal{B}^S.\Phi(p)$ if $\rho = o$. Given a fragment f,

$\mathcal{B}^S(f)$ denotes the SPBF for f. Given the definition of an SPBF, we adapt the general definition of a distributed index (Definition 20) to the setup of LOTHBROK in [9]. In other words, [9] defines an SPBF index as a structure that maps from star patterns to the set of fragments that may contain all the constants in the star pattern on the corresponding positions, and the fragments to the nodes that store the fragments. Given a star-shaped BGP P and a fragment f, [9] defines $relevantFragment(P, f)$ as a function that returns `true` iff $\forall t = (s, p, o) \in P$, $s \in V$ or $s \sqsubseteq^s \mathcal{B}^S(f)$, $p \in V$ or $p \sqsubseteq^p \mathcal{B}^S(f)$, and $o \in V$ or $o \sqsubseteq \mathcal{B}^S(f).\Phi(p)$, or `false` otherwise. Formally, an SPBF index is defined in [9] as follows:

Definition 36 (SPBF Index – reproduced from [9])
Let n be a node and $\mathcal{N}$ be the set of nodes within n's local view of the network, $\mathcal{P}$ be the set of all possible star patterns, and $\mathcal{F}$ be the set of fragments stored by at least one node in $\mathcal{N}$. The *SPBF index* on n is a tuple $I_n^S = (\upsilon, \eta)$ with $\upsilon : \mathcal{P} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^{\mathcal{N}}$. $\upsilon(P)$ returns the set of fragments F such that $\forall f \in F$, $relevantFragment(P, f) =$ `true`. $\eta(f)$ returns the set of nodes N such that $f \in n_i.\Gamma$ (Definition 10), $\forall n_i \in N$ and $n_i \in \mathcal{N}$.

Likewise, we extend the notion of an index slice (Definition 22), by defining an *SPBF slice* as follows [9]:

Definition 37 (SPBF Slice – reproduced from [9])
Let f be a fragment. The *SPBF slice* describing f is a tuple $s_f^S = (\upsilon', \eta')$ where $\upsilon'(P)$ returns $\{f\}$ if and only if $relevantFragment(P, f) =$ `true`, and $\eta'(f)$ returns the set of all nodes that contain f in their local datastore.

Given a fragment f, the SPBF slice describing f is the SPBF obtained from f and is denoted by $s^S(f)$. For instance, $s^S(f_1)$ in the previous example is the SPBF visualized in Figure 22a.

7.3 Query Optimization

Query optimization in LOTHBROK consists of several steps. First, LOTHBROK builds a *compatibility graph* that describes which relevant fragments produce join results for the joins in the query. Then, based on the compatibility graph and cardinality estimations, LOTHBROK builds a *query execution plan* using Dynamic Programming (DP). In this section, we summarize the query optimization strategy used by LOTHBROK by reproducing and adapting the definitions, formulas, algorithms, and examples from [9].

The output of the query optimization step is a query execution plan that specifies which subqueries can be processed in parallel (i.e., different branches in the compatibility graph), which subqueries are delegated to which nodes, and the join order. This is formally defined in [9] as follows:

Definition 38 (Query Execution Plan – reproduced from [9])
A *query execution plan* Π consists of the execution plan and the node that processes the plan, called a *delegation*. A query execution plan can be one of four types:

- *Join* $\Pi = \Pi_1 \bowtie^n \Pi_2$ where Π_1 and Π_2 are two (sub)plans and n is the node the join is delegated to.
- *Cartesian product* $\Pi = \Pi_1 \times^n \Pi_2$ where Π_1 and Π_2 are two (sub)plans and n is the node the Cartesian product is delegated to.
- *Union* $\Pi = \Pi_1 \cup^n \Pi_2$ where Π_1 and Π_2 are two (sub)plans and n is the node the union is delegated to.
- *Selection* $\Pi = [[P]]^n_f$ where P is a star pattern, f is the fragment that P is processed over, and n is the node the selection is delegated to.

Since unions are not executed by a specific node, instead the results of processing each subplan in the union are transferred to the nodes that process subsequent subqueries, like [9], we omit delegations from unions in the remainder of this section. Furthermore, as in [9], we assume that execution plans are left-deep, i.e., the right side of any join can only either be a selection or a union of selections.

Example 12 (Query execution plan – adapted from [9])
Consider the query execution plan $\Pi = (([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$ (Figure 28g). Π specifies that the join $[[P_2]]_{f_4} \bowtie [[P_1]]_{f_1}$ is delegated to n_2 and processed in parallel with $[[P_2]]_{f_3} \bowtie [[P_1]]_{f_2}$ on n_3, the result of which is transferred to n_1 and joined with $[[P_3]]_{f_5}$.

In the remainder of this section, we first detail source selection using compatibility graphs, after which we detail how Lothbrok uses the SPBF indexes to provide cardinality estimations. Last, we detail the cost function used in the DP algorithm to obtain the query execution plan.

Fragment and Source Selection

The first step in query optimization in Lothbrok is source selection by exploiting the information about fragment compatibility for a given query made possible by the SPBF indexes. Two fragments are said to be *compatible* for a given BGP if the intersection of the corresponding SPBF partitions is non-empty [9]. To consider fragment compatibility, Lothbrok first builds a *compatibility graph* in which nodes are the relevant fragment for the query and edges describe the compatible ones. In the following, we summarize how

LOTHBROK performs source selection by reproducing the definitions and algorithms from [9].

Given an SPBF $\mathcal{B}^S$, star pattern P, and variable v, $\mathcal{B}(\mathcal{B}^S, P, v)$ denotes a function that returns the corresponding SPBF partition in $\mathcal{B}^S$, i.e., $\mathcal{B}^S.\mathcal{B}_s$ if v is the subject in P, or $\mathcal{B}^S.\Phi(p)$ if v is the object with predicate v ($(s, p, v) \in P$) [9]. Compatibility graphs are formally defined in [9] as follows:

Definition 39 (Compatibility Graph – reproduced from [9])
Given an SPBF index I^S and a BGP P, the *compatibility graph* G^C of P over I^S is a tuple $G^C(P, I^S) = (F, C)$ such that $\forall P_1, P_2 \in \mathcal{S}(P)$ where $vars(P_1) \cap vars(P_2) \neq \emptyset$ and $\forall v \in vars(P_1) \cap vars(P_2)$, it is the case that $\forall f_1 \in I^S.v(P_1), f_2 \in I^S.v(P_2)$ where $\mathcal{B}(\mathcal{B}^S(f_1), P_1, v) \cap \mathcal{B}(\mathcal{B}^S(f_2), P_2, v) \neq \emptyset$, $(f_1, f_2) \in C$ and $f_1, f_2 \in F$. Furthermore, $\forall P' \subseteq P$ where $vars(P') \cap vars(P - P') = \emptyset$ (i.e., for Cartesian products), it is the case that $\forall f_1 \in F$ such that $f_1 \in I^S.v(P_1)$ for some $P_1 \in \mathcal{S}(P')$ and $\forall f_2 \in F$ such that $f_2 \in I^S.v(P_2)$ for some $P_2 \in \mathcal{S}(P - P')$, $(f_1, f_2) \in C$.

Algorithm 5 defines the G^C function that builds the compatibility graph of a BGP P given an SPBF index I^S using `buildBranch` in line 15. In the following, we will explain the algorithm by explaining the step-by-step process of building the compatibility graph for query Q in the running example (Figure 21a). The intermediate compatibility graph at each step of this process are visualized in Figure 23.

Example 13 (Building the compatibility graph – adapted from [9])
Consider query Q in Figure 21a and the characteristic sets in Figure 21b. Building the compatibility graph for Q, the first step in line 2 is to select the star pattern in P with the lowest estimated cardinality (cardinality estimation is detailed later in this section); assume in this example that P_2 is the star pattern with the lowest cardinality. Furthermore, assume that f_1 is compatible with $\{f_4, f_5\}$, and f_2 is compatible with $\{f_3, f_5\}$.

The for loop in lines 5-7 iterates through each relevant fragment to P_2, i.e., $\{f_3, f_4\}$; for each of these fragments, the `buildBranch` function is called in line 6 in order to build the subgraph for the current fragment. In the first iteration (i.e., for f_3), the `buildBranch` function is thus called with the parameters $P = P_1 \cup P_3$, $f = f_3$, and $P' = P_2$.

Since, in this case, P_1 joins with P_2, the for loop in line 22 iterates through all compatible fragments to f_3 for the given join, in this case $\{f_2\}$. Line 23 thus makes a recursive call with the parameters $P = P_3$, $f = f_2$, and $P' = P_1$. Here, since P_3 joins with P_1, the for loop in line 22 iterates through the compatible fragments to f_2 for the given join, i.e., $\{f_5\}$, and makes the recursive call in line 23 with parameters $P = \emptyset$, $f = f_5$, and $P' = P_3$.

Algorithm 5 Compute Compatibility Graph – adapted from [9]

Input: A BGP $P = P_1 \cup \cdots \cup P_n$; an SPBF index $I^S = (v, \eta)$
Output: A compatibility graph G^C

1: **function** $G^C(P, I^S)$
2: $P' \leftarrow P_k$ where $P_k \in \mathcal{S}(P)$ and $card_B(P_k) \leq card_B(P_j) \forall P_j \in \mathcal{S}(P)$;
3: $P_e \leftarrow P'$;
4: $F, C \leftarrow \emptyset$;
5: **for all** $f \in I^S.v(P')$ **do**
6: $G^C_e \leftarrow$ `buildBranch`$(P - P', I^S, f, P', P_e)$;
7: $F \leftarrow F \cup G^C_e.F$; $C \leftarrow C \cup G^C_e.C$;
8: **if** $P - P_e \neq \emptyset$ **then**
9: $G^C_e \leftarrow \mathtt{G}^C(P - P_e, I^S)$;
10: **if** $G^C_e = G^C_\emptyset$ **then return** $G^C_\emptyset$
11: **for all** $f_1 \in F, f_2 \in G^C_e.F$ **do**
12: $C \leftarrow C \cup \{(f_1, f_2)\}$;
13: $F \leftarrow F \cup G^C_e.F$; $C \leftarrow C \cup G^C_e.C$;
14: **return** (F, C);
15: **function** *buildBranch*(P, I^S, f, P', P_e)
16: **if** $P = \emptyset$ **or** $\forall P'' \in \mathcal{S}(P) : vars(P') \cap vars(P'') = \emptyset$ **then**
17: **return** $(\{f\}, \emptyset)$;
18: $F, C \leftarrow \emptyset$;
19: **for all** $P'' \in \mathcal{S}(P)$ s.t. $vars(P') \cap vars(P'') \neq \emptyset$ **do**
20: $P'_e \leftarrow P_e \cup P''$;
21: $V \leftarrow vars(P') \cap vars(P'')$;
22: **for all** $f' \in I^S.v(P'')$ s.t. $\forall v \in V : \mathcal{B}(\mathcal{B}^S(f), P', v) \cap \mathcal{B}(\mathcal{B}^S(f'), P'', v) \neq \emptyset$ **do**
23: $G^C_e \leftarrow$ `buildBranch`$(P - P'', I^S, f', P'', P'_e)$;
24: **if** $G^C_e \neq G^C_\emptyset$ **then**
25: $F \leftarrow F \cup G^C_e.F \cup \{f\}$; $C \leftarrow C \cup G^C_e.C \cup \{(f, f')\}$;
26: $P_e \leftarrow P_e \cup P'_e$;
27: **return** (F, C);

In this call to `buildBranch`, since $P = \emptyset$, the function returns in line 17 the compatibility graph $(\{f_5\}, \emptyset)$ visualized in Figure 23a. Since this graph is non-empty, it is added to the output graph in line 25 together with f_2 (since $f = f_2$ in the current iteration of `buildBranch`) and an edge between f_2 and f_5, as visualized in Figure 23b, and returned. Again, since this graph is non-empty, it is added to the output graph in line 25 along with f_3 (since $f = f_3$ in the current iteration) and an edge between f_2 and f_3,

resulting in the compatibility graph visualized in Figure 23c. This graph is returned in line 27.

Following the same procedure in the next iteration of the for loop in line 5, we call the `buildBranch` function with $f = f_4$ and build the compatibility graph by first adding f_5 (Figure 23d), then f_1 along with an edge between f_1 and f_5 (Figure 23e), following by f_4 and an edge between f_4 and f_1 (Figure 23f). Since the query contains no Cartesian products, the two subgraphs are merged into the compatibility graph shown in Figure 23g (line 7) and returned in line 14.

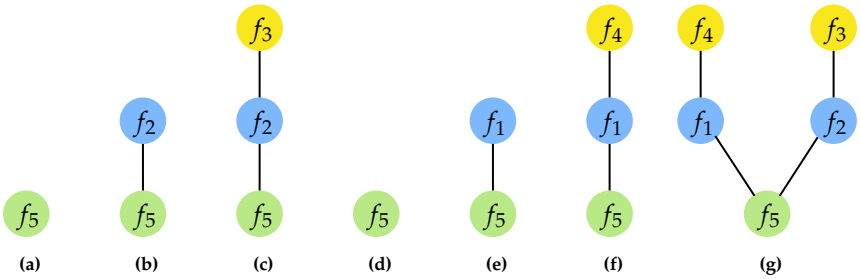


Fig. 23: Recursively building the compatibility graph for the query in Figure 21a by applying Algorithm 5 resulting in $G^C(Q, I^S_{n_1})$ (reproduced from [9]).

Cardinality Estimation

Lothbrok builds the query execution plan for a query using Dynamic Programming (DP). Besides considering the compatibility graph, the DP algorithm also uses a cost function that considers the estimated data transfer overhead for a specific execution plan. To obtain this cost, we use the estimated cardinality of an execution (sub)plan. Furthermore, Since SPBFs include partitioned bitvectors describing the subject and objects of a fragment (Definition 35), we can estimate the occurrences of each characteristic set similar to [67, 70] by estimating the cardinality of the corresponding SPBF partitions. In the following, we summarize the cardinality estimation technique used by Lothbrok by reproducing the equations and descriptions from [9].

Given a partitioned bitvector $\mathcal{B}$ and $\hat{b} \in \mathcal{B}.\hat{B}$, let $t(\hat{b})$ return the number of bits in $\hat{b}$ that are set to 1. The estimated cardinality of $\mathcal{B}$, denoted $card^P(\mathcal{B})$, is the sum of the estimated cardinality of each bitvector in $\mathcal{B}$ as defined in [6, 71], and is formally defined in [9] as follows:

$$card^P(\mathcal{B}) = \sum_{\hat{b} \in \mathcal{B}.\hat{B}} \frac{ln(1 - t(\hat{b})/|\hat{b}|)}{|\mathcal{B}.H| \cdot ln(1 - 1/|\hat{b}|)} \tag{19}$$

Example 14 (Partitioned bitvector cardinality – adapted from [9])
Consider the SPBF for f_4 in the running example (Figure 21), $\mathcal{B}^S(f_4)$ in Figure 22b, and assume that $|\mathcal{B}^S(f_4).H| = 5$ and $|\hat{b}| = 20000$ for all $\hat{b} \in \mathcal{B}^S(f_4).\hat{B}$. When estimating the cardinality of the partition $\mathcal{B}^S(f_4).\Phi(\texttt{dbo} : \texttt{capital})$ that has two partitions, `dbr` and `dbp`, assume that the number of set bits in `dbr` is 736 and the number of set bits in `dbp` is 249; applying Equation 19 yields the following equation:

$$card^P(\mathcal{B}^S(f_4).\Phi(\texttt{dbo:capital})) = \frac{ln(1-736/20000)}{5 \cdot ln(1-1/20000)} + \frac{ln(1-249/20000)}{5 \cdot ln(1-1/20000)} \approx \frac{-0.0375}{-0.00025} + \frac{-0.0125}{-0.00025} \approx 150 + 50 \approx 200$$

Table 3 shows the cardinality of each partitioned bitvector in the running example.

Table 3: Estimated cardinalities for the SPBFs $\mathcal{B}^S(f_1)$, $\mathcal{B}^S(f_2)$, $\mathcal{B}^S(f_3)$, and $\mathcal{B}^S(f_4)$ for the running example in Figure 20 (reproduced from [9])

Partitioned Bitvector	$card^P$	Partitioned Bitvector	$card^P$
$\mathcal{B}^S(f_1).\mathcal{B}_s$	1000	$\mathcal{B}^S(f_3).\mathcal{B}_s$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:author})$	5000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:capital})$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality})$	1000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:currency})$	150
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:deathDate})$	1000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:population})$	100
$\mathcal{B}^S(f_2).\mathcal{B}_s$	2000	$\mathcal{B}^S(f_4).\mathcal{B}_s$	200
$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:author})$	3000	$\mathcal{B}^S(f_4).\Phi(\texttt{dbo:capital})$	200
$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality})$	2000	$\mathcal{B}^S(f_4).\Phi(\texttt{dbo:currency})$	500
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_3).\mathcal{B}_s$	0	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_3).\mathcal{B}_s$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s$	50	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s$	0
$\mathcal{B}^S(f_5).\mathcal{B}_s$	8000	$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:author}) \cap \mathcal{B}^S(f_5).\mathcal{B}_s$	500
$\mathcal{B}^S(f_5).\Phi(\texttt{dbo:publisher})$	8000	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:author}) \cap \mathcal{B}^S(f_5).\mathcal{B}_s$	1000
$\mathcal{B}^S(f_5).\Phi(\texttt{dbo:language})$	9000		

Given a star pattern P asking for the set of unique subject values described by the predicates in P (i.e., queries with the `DISTINCT` keyword) and a specific fragment f, the cardinality of P over f is the cardinality of the subject partition in $\mathcal{B}^S(f)$, formalized in [9] as follows:

$$card_D(P, f) = card^P(\mathcal{B}^S(f).\mathcal{B}_s) \tag{20}$$

For queries that do not include the `DISTINCT` keyword, we account for duplicates by also considering the number of triples, on average, for each

predicate in P that each subject is associated with, i.e., the predicate occurrences [9]. Let $preds(P)$ denote the predicates in P. Then, the cardinality of P over f is defined in [9] as follows:

$$card_S(P, f) = card_D(P, f) \cdot \prod_{p_i \in preds(P)} \frac{card^P(\mathcal{B}^S(f).\Phi(p_i))}{card^P(\mathcal{B}^S(f).\mathcal{B}_s)} \tag{21}$$

In the remainder of this section, as in [9], we will refer to *card* as being equivalent to $card_D$ for queries with the `DISTINCT` keyword and $card_S$ for queries without. The cardinality of a star pattern P over a node n's SPBF index is the aggregated cardinality over each relevant fragment, formally defined in [9] as follows:

$$card_n(P) = \sum_{f \in I_n^S.\eta(P)} card(P, f) \tag{22}$$

Example 15 (Cardinality estimation (`DISTINCT`) – adapted from [9])
Consider P_1 in Figure 21a and the SPBF partition cardinalities in Table 3. Given the `DISTINCT` keyword, applying Equation 22 yields the following equation: $card_{n_1}(P_1) = 1000 + 2000 = 3000$ (Equation 20), visualized in Figure 24.

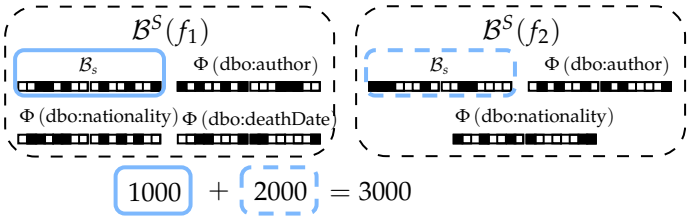


Fig. 24: Estimating the cardinality of P_1 with the `DISTINCT` modifier (reproduced from [9]).

Example 16 (Cardinality estimation – adapted from [9])
Consider P_1 in Figure 21a and the SPBF partition cardinalities in Table 3. Applying Equation 22 without the `DISTINCT` keyword yields the following equation: $card_{n_1}(P_1) = 1000 \cdot (5000/1000) \cdot (1000/1000) + 2000 \cdot (3000/2000) \cdot (2000/2000) = 5000 + 3000 = 8000$ (Equation 21), visualized in Figure 25.

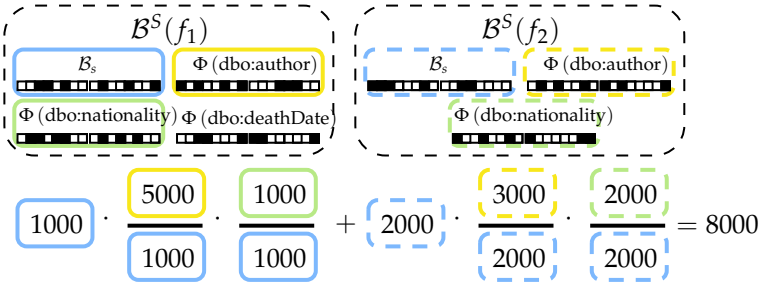


Fig. 25: Estimating the cardinality of P_1 without the `DISTINCT` modifier (reproduced from [9]).

To estimate the cardinality of arbitrary (sub)queries, we extend the frameworks presented above and in [34, 67] to estimate the cardinality of a query execution plan, and reproduce the equations from [9] in the following.

The cardinality of Cartesian products, unions, and selections, is straightforward and defined in [9] as follows. For Cartesian products, it is the multiplication of the cardinality of the operands, for unions it is the addition of the cardinality of the operands, and the cardinality of selections is found using Equation 22. As such, the cardinality of a query execution plan is given in [9] by the following equation:

$$card(\Pi) = \begin{cases} card(\Pi_1) \cdot card(\Pi_2), & \text{if } \Pi = \Pi_1 \times^n \Pi_2 \\ card(\Pi_1) + card(\Pi_2), & \text{if } \Pi = \Pi_1 \cup \Pi_2 \\ card(P, f), & \text{if } \Pi = [[P]]^n_f \\ card^{\bowtie}(\Pi_1 \bowtie^n \Pi_2), & \text{if } \Pi = \Pi_1 \bowtie^n \Pi_2 \end{cases} \tag{23}$$

The function $card^{\bowtie}(\Pi_1 \bowtie^n \Pi_2)$ exploits the fact that the right side of a join can only be a selection or a union of selections, and is formally defined in [9] as follows:

$$card^{\bowtie}(\Pi_1 \bowtie^n \Pi_2) = \begin{cases} card(\Pi_1 \bowtie^n \Pi_2') + \\ \qquad card(\Pi_1 \bowtie^n \Pi_2''), & \text{if } \Pi_2 = \Pi_2' \cup \Pi_2'' \\ card^{\bowtie}(\Pi_1, P, f), & \text{if } \Pi_2 = [[P]]^{n_1}_f \end{cases} \tag{24}$$

The $card^{\bowtie}(\Pi, P, f)$ function estimates the cardinality of the join for a particular selection on the right side of the join, $[[P]]_f$, similar to characteristic pairs [34]. As in [9], we consider the estimated cardinality of Π and the selectivity of the join; however if there is more than one join variable, we only consider the most selective join variable to avoid a huge overestimation due to the possible correlation between the join variables. Let $S(\Pi, P)$ denote the star patterns in Π that join with P, $F(\Pi, f)$ denote the fragments in Π that join with f, and $v(P_1, P_2) = \{v \mid v \in vars(P_1) \cap vars(P_2)\}$, i.e., the set of join

variables [9]. Then, given the DISTINCT keyword, the cardinality of a join between a plan Π and a selection $[[P]]_f$ is computed in [9] using the following equation:

$$card_D^{\bowtie}(\Pi, P, f) = card(\Pi) \cdot \min_{P' \in S(\Pi,P) \wedge v \in v(P,P')} \frac{\sum_{f' \in F(\Pi,f)} card^P(\mathcal{B}(\mathcal{B}^S(f), P, v) \cap \mathcal{B}(\mathcal{B}^S(f'), P', v))}{\sum_{f' \in F(\Pi,f)} card^P(\mathcal{B}(\mathcal{B}^S(f'), P', v))} \quad (25)$$

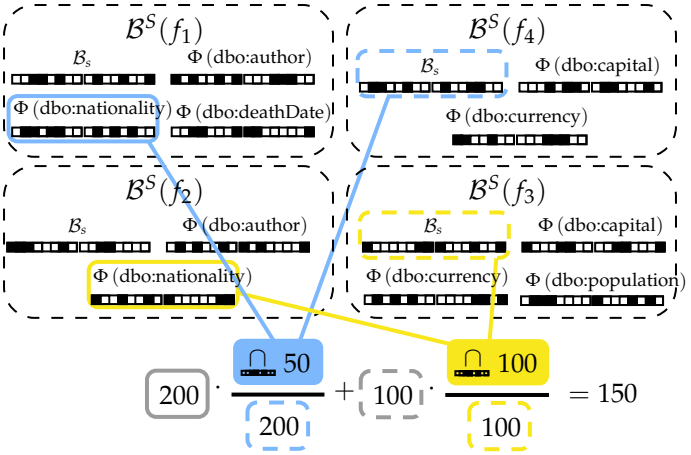


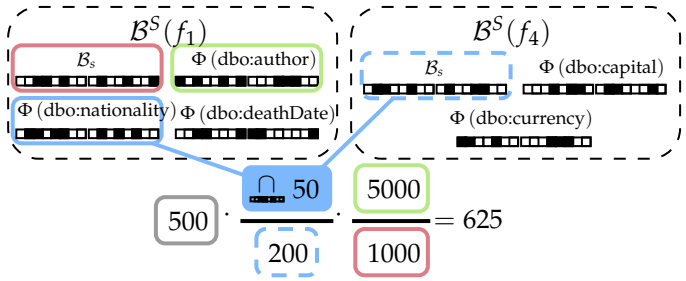
Fig. 26: Estimating the cardinality of $\Pi = ([[P_2]]_{f_4}^{n_2} \bowtie^{n_2} [[P_1]]_{f_1}^{n_2}) \cup ([[P_2]]_{f_3}^{n_3} \bowtie^{n_3} [[P_1]]_{f_2}^{n_3})$ with the DISTINCT keyword (reproduced from [9]).

Example 17 (Join cardinality estimation (DISTINCT) – adapted from [9]) Consider computing $card(\Pi)$ where $\Pi = ([[P_2]]_{f_4}^{n_2} \bowtie^{n_2} [[P_1]]_{f_1}^{n_2}) \cup ([[P_2]]_{f_3}^{n_3} \bowtie^{n_3} [[P_1]]_{f_2}^{n_3})$ (i.e., the execution plan in Figure 28d). Since Π is a union, we compute the cardinality of $\Pi_1 = [[P_2]]_{f_4}^{n_2} \bowtie^{n_2} [[P_1]]_{f_1}^{n_2}$ and $\Pi_2 = [[P_2]]_{f_3}^{n_3} \bowtie^{n_3} [[P_1]]_{f_2}^{n_3}$ separately and aggregate the result. This yields the following formula: $card(\Pi) = 200 \cdot (50/200) + 100 \cdot (100/100) = 150$ (visualized in Figure 26).

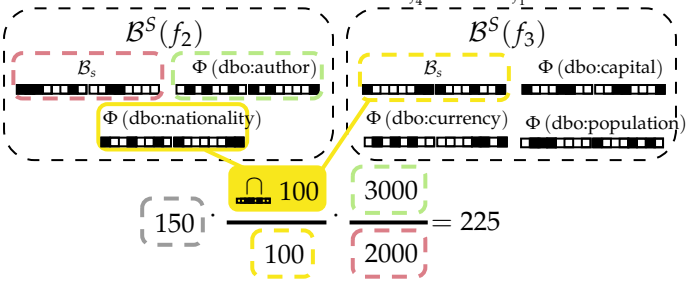
For queries without the DISTINCT keyword, we consider in [9] the average number of predicate occurrences in f for each triple pattern in P that does not join on the object position with any star pattern in Π. This is formally defined as follows [9]:

$$card_S^{\bowtie}(\Pi, P, f) = card_D^{\bowtie}(\Pi, P, f) \cdot \prod_{p \in preds(P): (s,p,o) \in P \wedge o \notin v(P,P') \forall P' \in S(\Pi,P)} \left(\frac{card^P(\mathcal{B}^S(f).\Phi(p))}{card^P(\mathcal{B}^S(f).\mathcal{B}_s)} \right) \quad (26)$$

Example 18 (Join cardinality estimation – adapted from [9])
Consider again $\Pi = ([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$ in Figure 28d. Using Equation 26 yields the following formula: $card(\Pi) = 500 \cdot (50/200) \cdot (5000/1000) + 150 \cdot (100/100) \cdot (3000/2000) = 625 + 225 = 850$ (visualized in Figure 27).



(a) Cardinality of $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$



(b) Cardinality of $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$

Fig. 27: Estimating the cardinality of Π in Figure 28d without the `DISTINCT` modifier for (a) $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$ and (b) $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$. The output of Equation 23 is thus the sum of the two formulas (reproduced from [9]).

Optimizing Query Execution Plans

To compute the optimal query execution plan, LOTHBROK considers the cardinality estimations and compatibility graph described above, and applies

Dynamic Programming (DP) to obtain the execution plan. The cost function used by the DP algorithm, however, considers the locality of the data by estimating the number of intermediate results to be transferred over the network for a given delegation strategy based on the locality of the data.

Algorithm 6 Compute transfer cost – adapted from [9]

Input: A query execution plan Π; a node n
Output: The estimated transfer cost *cost*

1: **function** *transferCost*(Π,n)
2: $cost \leftarrow 0$;
3: **if** $\Pi = [[P]]_f^{n_i}$ **then**
4: **if** $n \neq n_i$ **then** $cost \leftarrow card(P, f)$;
5: **else if** $\Pi = \Pi_1 \cup \Pi_2$ **then**
6: $cost \leftarrow \texttt{transferCost}(\Pi_1, n) + \texttt{transferCost}(\Pi_2, n)$;
7: **else if** $\Pi = \Pi_1 \times^{n_i} \Pi_2$ **then**
8: $cost \leftarrow \texttt{transferCost}(\Pi_1, n_i) + \texttt{transferCost}(\Pi_2, n_i)$;
9: **if** $n_i \neq n$ **then** $cost \leftarrow cost + card(\Pi)$;
10: **else if** $\Pi = \Pi_1 \bowtie^{n_i} \Pi_2$ **then**
11: **if** $\Pi_2 = \Pi_2' \cup \Pi_2''$ **then**
12: $cost \leftarrow \texttt{transferCost}(\Pi_1 \bowtie^{n_i} \Pi_2', n) + \texttt{transferCost}(\Pi_1 \bowtie^{n_i} \Pi_2'', n)$;
13: **else if** $\Pi_2 = [[P]]_f^{n_j}$ **then**
14: $cost \leftarrow \texttt{transferCost}(\Pi_1, n_i)$;
15: **if** $n_i \neq n_j$ **then** $cost \leftarrow cost + card^{\bowtie}(\Pi_1, P, f)$;
16: **if** $n \neq n_i$ **then** $cost \leftarrow cost + card(\Pi)$;
17: **return** *cost*;

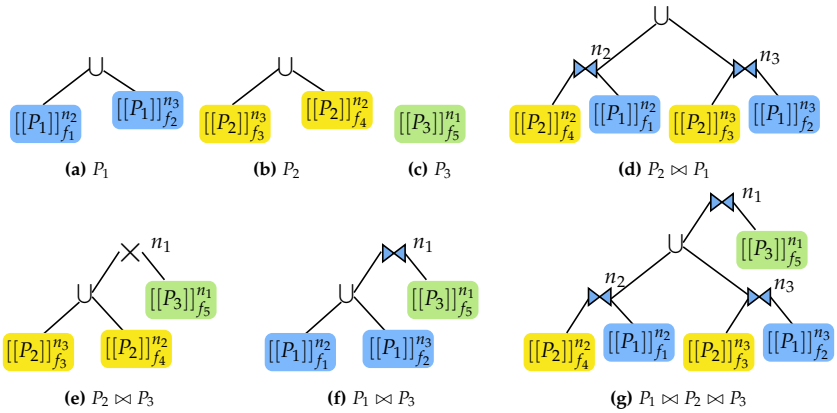
Algorithm 6 specifies the function that estimates the transfer cost. Once again, we shall explain the algorithm by guiding the reader through an example based on the running example.

Example 19 (Transfer cost – adapted from [9])
Consider the execution plan Π shown in Figure 28g processed by node n_1. To estimate the transfer cost of the delegations in Π, the `transferCost` function first enters the if statement on line 10 (since Π is a join) and the if statement on line 13 (since the right side is a selection). Since n_1 is specified in the delegation, i.e., it is processed locally, the function only considers the transfer cost on the left side and makes a recursive call on line 14.

In this call, for $\Pi = ([[P_2]]_{f_4}^{n_2} \bowtie^{n_2} [[P_1]]_{f_1}^{n_2}) \cup ([[P_2]]_{f_3}^{n_3} \bowtie^{n_3} [[P_1]]_{f_2}^{n_3})$, the function enters the if statement on line 5 and makes one recursive call for each (sub)plan in the union.

In the first recursive call, i.e., for $\Pi = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$, the function once again enters the if statements on line 10 and 13. Here, since the join is delegated to n_2 and $n_2 \neq n_1$, we consider both the cost of the left side (line 14) and the cardinality of Π (line 16); since the left side ($([[P_2]]^{n_2}_{f_4}$) is also delegated to n_2, the transfer cost of the left side is 0, while the cardinality of Π is 625 (Figure 27a).

Following the same procedure for $\Pi = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$, the transfer cost of the union is thus $625 + 225 = 850$, which is also the final result since the top-most join is delegated to n_1, i.e., the plan incurs transferring the 850 results from the two subplans in the union from n_2 and n_3 to n_1.



(a) P_1 **(b)** P_2 **(c)** P_3 **(d)** $P_2 \bowtie P_1$

(e) $P_2 \bowtie P_3$ **(f)** $P_1 \bowtie P_3$ **(g)** $P_1 \bowtie P_2 \bowtie P_3$

Fig. 28: Best query execution plan for each subquery in the DP table (reproduced from [9]).

In [9], the cost of processing a query execution plan Π on a node n is defined as the transfer cost plus the cardinality estimation as follows:

$$cost_n(\Pi) = transferCost(\Pi, n) + card(\Pi) \tag{27}$$

Lothbrok uses the cost function in Equation 27, to estimate the cost of all possible delegation plans and applies DP to obtain the one that has the lowest overall cost [9].

Example 20 (DP cost – adapted from [9])
Table 4 shows the (partial) DP table entries for the lowest costing execution plan for each subquery in Q (Figure 21a) on node n_1 in the running example. These plans are visualized in Figure 28.

Table 4: Entries in the DP table for query Q (reproduced from [9])

Subquery	Execution Plan	Cardinality	Cost
P_1	$[[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}$	8,000	8,000
P_2	$[[P_2]]^{n_3}_{f_3} \cup [[P_2]]^{n_2}_{f_4}$	650	650
P_3	$[[P_3]]^{n_1}_{f_5}$	9,000	9,000
$P_2 \bowtie P_1$	$([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$	**850**	1,700
$P_2 \bowtie P_3$	$([[P_2]]^{n_3}_{f_3} \cup [[P_2]]^{n_2}_{f_4}) \times^{n_1} [[P_3]]^{n_1}_{f_5}$	5,850,000	5,850,650
$P_1 \bowtie P_3$	$([[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$	1,688	9,688
$P_2 \bowtie P_1 \bowtie P_3$	$(([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$	154	1,004

7.4 Query Execution

In this section, we describe how Lothbrok executes a query execution plan by reproducing the definitions and algorithms from [9].

Given a query execution plan Π over a BGP P and compatibility graph G^C, Lothbrok processes P by processing the operations specified in Π and delegating joins and Cartesian products to the nodes specified by delegations in Π [9]. To formalize how star patterns in Π are processed over the corresponding fragments, we build on the selector function from Definition 33 and define the star pattern-based selector function as follows [9]:

Definition 40 (Star Pattern Selector Function – reproduced from [9])
Given a node n, a star pattern P, and a finite set of distinct solution mappings Ω, the star pattern-based selector function for P and Ω, denoted $s_{(P,\Omega)}$ is for every fragment f in n's local datastore defined as follows.

$$s_{(P,\Omega)}(f) = \begin{cases} \{t \in T \mid T \subseteq f \wedge T[P]\} & \text{if } \Omega = \emptyset \\ \{t \in T \mid T \subseteq f \wedge \exists \mu \in [[P]]_f, \mu' \in \Omega : \\ \qquad \mu[P] = T \wedge \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

In line with [4, 7, 38], we apply pagination of the results; however, for ease of presentation, we assume in the remainder of this section that all results can fit into a single page [9]. Given a star pattern P, a node n, a fragment f, and a finite set of solution mappings Ω, $sel_n(f, P, \Omega)$ denotes in [9] the result of invoking $s_{(P,\Omega)}(f)$ on n.

Algorithm 7 defines the `evaluatePlan` function that executes a query execution plan. First, if the execution plan is a Cartesian product, the function processes each side concurrently in lines 3-4 and combines the result as the Cartesian product of the two sets of intermediate results (line 5). If instead it is a join plan, the function processes the left side first in line 7 and uses the intermediate bindings from that when processing the right side in line 8. However, if the plan is a union, the function processes each side concurrently in lines 10-11 and combines those result in line 12. Last, if the plan is a selection, the function calls the selector function on the node specified by the

delegation in line 14 and joins the results with the intermediate results found from previously evaluated star patterns in line 15.

Algorithm 7 Evaluate an execution plan – adapted from [9]

Input: A join plan Π; a node n; a set of solution mappings Ω
Output: A set of solution mappings Ω

1: **function** *evaluatePlan*(Π,n,$\Omega = \{\emptyset\}$)
2: **if** $\Pi = \Pi_1 \times^{n_i} \Pi_2$ **then**
3: $\Omega_1 \leftarrow$ `evaluatePlan`(Π_1, n_i, Ω);
4: $\Omega_2 \leftarrow$ `evaluatePlan`(Π_2, n_i, Ω);
5: $\Omega \leftarrow \Omega_1 \times \Omega_2$;
6: **else if** $\Pi = \Pi_1 \bowtie^{n_i} \Pi_2$ **then**
7: $\Omega \leftarrow$ `evaluatePlan`(Π_1, n_i, Ω);
8: $\Omega \leftarrow$ `evaluatePlan`(Π_2, n_i, Ω);
9: **else if** $\Pi = \Pi_1 \cup \Pi_2$ **then**
10: $\Omega_1 \leftarrow$ `evaluatePlan`(Π_1, n, Ω);
11: $\Omega_2 \leftarrow$ `evaluatePlan`(Π_2, n, Ω);
12: $\Omega \leftarrow \Omega_1 \cup \Omega_2$;
13: **else if** $\Pi = [[P]]^{n_i}_f$ **then**
14: $\phi \leftarrow sel_{n_i}(f, P, \Omega)$;
15: $\Omega \leftarrow \Omega \bowtie \{\mu \mid dom(\mu) = vars(P) \text{ and } \mu[P] \in \phi\}$;
16: **return** Ω;

Example 21 (Processing a query execution plan – adapted from [9])
Consider the query execution plan Π shown in Figure 28g processed by node n_1 (Figre 20). Since this execution plan is a join, the function first makes a recursive call in line 7 with the plan $\Pi_1 = ([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$.

Since Π_1 is a union, the function then makes two concurrent recursive calls in lines 10-11 with the two subplans $[[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$ on n_2 and $[[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$ on n_3. This delegation is visualized in Figure 29a.

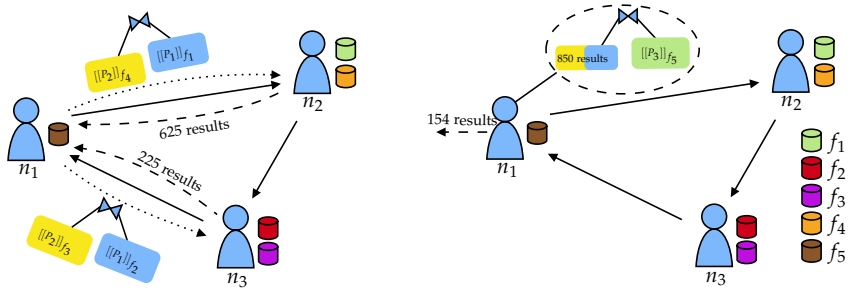
Node n_2, for $[[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$, makes a recursive call in line 7 for the plan $[[P_2]]^{n_2}_{f_4}$ for which it calls the local selector function (since n_2 is specified in the delegation) in line 14. The 500 results (cf. Table 3) are joined with the singleton set of bindings $\Omega = \{\emptyset\}$.

Upon receiving the 500 results in line 7, the function makes another recursive call in line 8 for the plan $[[P_1]]^{n_2}_{f_1}$, again calling its local selector in line 14. This results in 625 results to the subplan $[[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$.

While n_2 found the 625 results of processing $[[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$, n_3

found an additional 225 results for the subplan $[[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$ in the recursive call in line 11 following the same steps as n_2. This step is visualized in Figure 29b.

The 850 intermediate bindings found by processing $([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$ are used as Ω when processing the subplan $[[P_3]]^{n_1}_{f_5}$ using the local selector on n_1. This results in 154 results to Π.



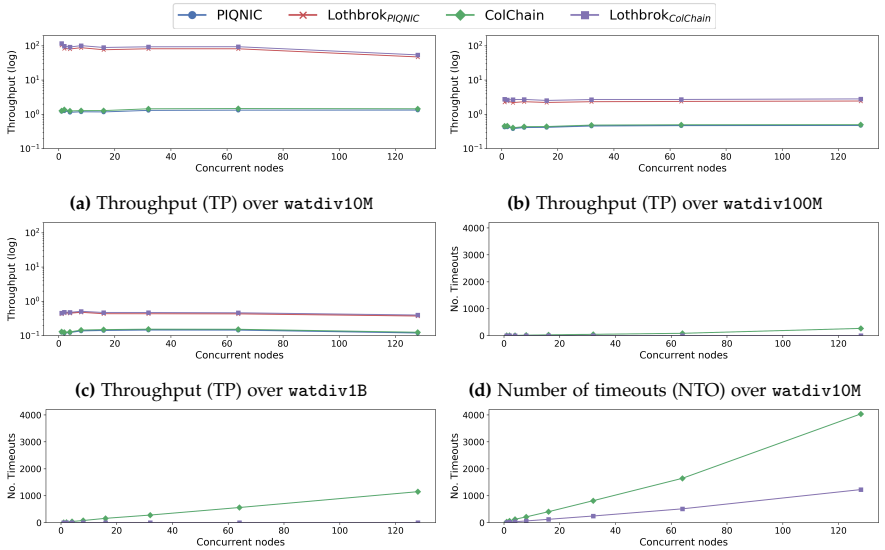
(a) Delegating subqueries to n_2 and n_3

(b) Processing $[[P_3]]_{f_5}$ locally on node n_1

Fig. 29: Processing Π in Figure 28g on n_1 by (a) delegating $[[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}$ to n_2 and $[[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2}$ to n_3 concurrently and (b) processing the join between these 850 results and $[[P_3]]_{f_5}$ locally on n_1 to achieve the 154 results (reproduced from [9]).

7.5 Evaluation and Discussion

The approaches presented outlined in Sections 4-6 take a first step in solving some of the major obstacles the Semantic Web faces today by increasing the availability and updatability of knowledge graphs on the Web. However, even so, they propose approaches that incur a significant communication overhead since nodes have to query information available on several different nodes when processing queries, decreasing performance overall. In Paper F [9], we therefore introduced Lothbrok, a novel approach to decreasing the communication overhead for processing queries in decentralized systems and, in doing so, increasing performance. Lothbrok does so by fragmenting the data based on characteristic sets [70] and introducing a novel indexing strategy, called Semantically Partitioned Bloom Filter (SPBF) indexes, that indexes the IRIs based on their position in the fragment. Furthermore, Lothbrok applies a query processing strategy that considers fragment compatibility, data locality, and cardinality estimations to delegate subqueries to other nodes such that the number of intermediate results transferred over the network is minimized.



(a) Throughput (TP) over `watdiv10M`

(b) Throughput (TP) over `watdiv100M`

(c) Throughput (TP) over `watdiv1B`

(d) Number of timeouts (NTO) over `watdiv10M`

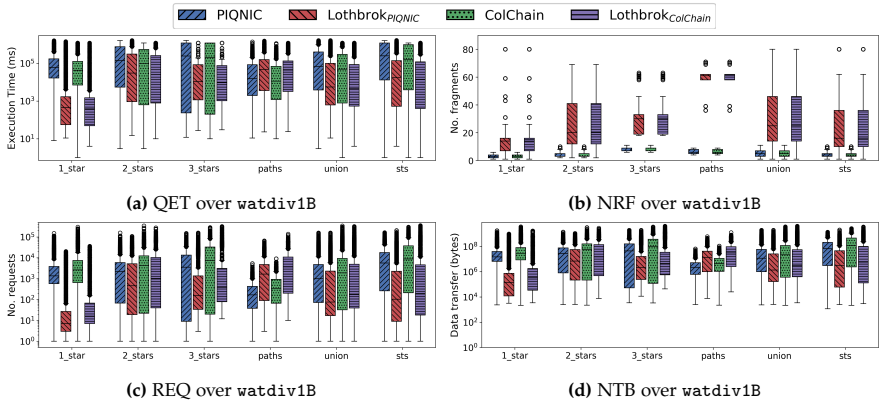
(e) Number of timeouts (NTO) over `watdiv100M`

(f) Number of timeouts (NTO) over `watdiv1B`

Fig. 30: Throughput (TP), and number of timeouts (NTO) for `watdiv-sts` over `watdiv10M`, `watdiv100M`, and `watdiv1B` (reproduced from [9]).

Thus, Lothbrok overcomes the last obstacle that we address in this thesis; ensuring efficient query processing performance over the decentralized knowledge graph systems presented thus far. In order to evaluate our approach, we implemented Lothbrok on top of both Piqnic and ColChain and ran experiments using the same WatDiv [13] datasets from Section 3 as well as the LargeRDFBench [77] data and queries on a network with 128 nodes. For queries over the WatDiv datasets, we used in [9] the `watdiv-sts` query load from Section 3 and used the WatDiv query template generator to generate query loads with 1-3 star patterns each, called the `watdiv-1_star`, `watdiv-2_star`, and `watdiv-3_star` query loads, as well as a query load consisting of path queries, called `watdiv-path`. Furthermore, we combine the aforementioned query loads into a single load spanning all the queries, called `watdiv-union`. Our full experimental evaluation is given in Paper F [9]; in the following, we will focus our discussion on the most interesting experimental results.

Figure 30 shows the query throughput, i.e., the number of executed queries per minute (Figure 30a-30c), and the number of queries that timed out (Figure 30d-30f) in logarithmic scale for each WatDiv dataset over `watdiv-sts` for an increasing number of nodes issuing queries concurrently. Clearly, Lothbrok has a significantly better performance across all configurations than both Piqnic and ColChain. This is most significant for `watdiv10M`



(a) QET over `watdiv1B`

(b) NRF over `watdiv1B`

(c) REQ over `watdiv1B`

(d) NTB over `watdiv1B`

Fig. 31: (a) Query execution time (QET), (b) number of relevant fragments (NRF), (c) number of requests (REQ), and (d) number of transferred bytes (NTB), over `watdiv1B` and the WatDiv star queries (reproduced from [9]).

where the increase in performance is up to two orders of magnitude; however, even for the largest dataset, `watdiv1B`, the increase is close to an order of magnitude. Furthermore, the performance of Lothbrok remains relatively stable for increasing network loads; even when all nodes in the network issue queries concurrently, the performance decreases only very little. Since Lothbrok increases performance, it is also able to answer more queries without timing out (in our experiments, the timeout was set to 1200 seconds, or 20 minutes); in fact, for `watdiv10M` and `watdiv100M`, Lothbrok experiences zero timeouts. Only for the largest dataset does Lothbrok experience a relatively small number of timeouts, albeit a much lower number than Piqnic and ColChain.

Figure 31 shows the execution time (Figure 31a), number of fragments relevant to each query after optimization (Figure 31b), the number of requests made between nodes when processing the queries (Figure 31c), and the number of transferred bytes between nodes (Figure 31d) for each WatDiv query load over `watdiv1B`. Like with the scalability experiments in Figure 30, Figure 31a shows a significant improvement on performance across most query loads for Lothbrok. Furthermore, the performance increase is more significant, the fewer star patterns the queries contain, since each star pattern in `watdiv-1_star` represent larger parts of the queries than in `watdiv-3_star`.

The only exception to this is `watdiv-paths`, where Lothbrok has slightly worse performance than Piqnic and ColChain. This is caused by Lothbrok having a significantly higher number of relevant nodes compared to Piqnic and ColChain (Figure 31b) leading to a larger number of fragments needed to be queried, while the `watdiv-path` queries are not compensated by the increased performance that the query optimization provides, like the

remaining query loads. Nevertheless, even for `watdiv-union`, we see a significant performance increase on average.

Last, we notice that LOTHBROK incurs a significantly lower network load than PIQNIC and COLCHAIN (Figure 31c-31d) for all query loads except `watdiv-path` (for the reasons specified above), reducing the communication overhead when processing queries. This is caused by LOTHBROK being able to process each star pattern with fewer requests compared to PIQNIC and COLCHAIN, even if there are more relevant fragments.

Overall, our experimental results show that LOTHBROK significantly reduces the communication overhead and, in doing so, improves performance by up to two orders of magnitude compared to PIQNIC and COLCHAIN. By implementing LOTHBROK on top of existing systems like COLCHAIN, we have thus shown that we can overcome some of the main obstacles that the Semantic Web faces today, i.e., data availability, scalability, and updatability, all while ensuring efficient query processing performance and high throughput.

8 Conclusions and Summary of Contributions

In conclusion, this thesis proposes novel techniques and robust methodologies that increase the availability, scalability, and updatability of knowledge graphs published on the Web while ensuring efficient access to them. As mentioned in Section 1.2, the six papers included in the thesis solve overall problem by addressing each issue individually with the goal of making knowledge graphs on the Web available, scalable, and updatable. In summary, the six papers included in this thesis make the following contributions:

- Paper A [18] introduces WISEKG; a client-server architecture that finds a novel, and in many cases better, balance between shifting some of the query processing load to the client while also exploiting the stronger server-side query processing capabilities when possible. To achieve this, WISEKG applies a cost model that dynamically determines whether it is most efficient to process each star-shaped subquery on the client or the server taking into account factors such as the data transfer cost and the current load on the server. A comprehensive experimental evaluation comparing WISEKG to state-of-the-art LDF systems shows that WISEKG significantly improves performance for high demanding workloads and that the cost model used by WISEKG improves the average workload completion time while answering more queries without timing out.
- Paper B [5] proposes a decentralized and unstructured P2P architecture, called PIQNIC, that increases the availability of the uploaded knowledge graphs by replicating the data across nodes in the P2P network. In PIQNIC, a set of homogeneous nodes in a network act as both clients

and servers and thus maintain a local datastore (a set of fragments) and a local view over the network (a set of neighboring nodes). As such, by replicating the data across multiple nodes, PIQNIC ensures that, even if the original uploader fails, the data is still accessible. PIQNIC implements three different query processing strategies, called `Single`, `Bulk`, and `Full`. Experimental results show that PIQNIC maintains a high availability of the data ($> 90\%$ availability) even when a third of the nodes in the network fails. Furthermore, the experimental results show that the `Bulk` query processing method significantly outperforms the other query processing strategies in all cases.

- Paper C [6] increases the query processing efficiency in unstructured P2P systems by introducing two novel indexing schemes that remove the need for flooding that network with requests and allow the nodes to determine which other nodes in the network contain relevant data to which parts of the query. First, *locational indexes* form the baseline indexes by matching each predicate in the query to the fragments that contain triples with the predicate. Building upon the baseline locational indexes, *PPBF* indexes represent the set of subjects and objects in each fragment as a partitioned bitvector; by checking whether or not the intersection of two such bitvectors is empty, nodes can prune fragments from the query processing plan that would not contribute to the overall query result. Experimental results over a large-scale benchmarking suite for federated systems confirm our hypothesis that such indexes significantly improve query processing performance by decreasing the huge network overhead that PIQNIC suffers from.
- Paper D [7] tackles the updatability problem by introducing COLCHAIN, a P2P system that allows users to collaborate on keeping the uploaded knowledge graphs up-to-date. To achieve this, COLCHAIN divides the entire network into smaller communities of nodes that share certain fragments and relies on community-wide consensus in order to allow any node within the community to propose updates to the fragments. Furthermore, COLCHAIN represents the history of updates to a knowledge graph as a *chain* of transactions (updates); this allows nodes to trace-back faulty updates to their origins and roll-back updates to an earlier point in time and process queries over them. Our experimental results show that COLCHAIN allows for efficient community-wide updates to knowledge graphs without incurring a significant cost on performance. Furthermore, the results show that COLCHAIN lets users efficiently process queries over earlier fragment versions.
- Paper E [8] demonstrates COLCHAIN as presented in Paper D [7] and provides a fully function COLCHAIN client with a graphical UI on top.

We demonstrate how users can navigate the fragments stored in communities they participate in and propose updates to any fragment. Furthermore, we show how users can vote on proposed updates to obtain community-wide consensus. Last, we demonstrate how queries can be processed over earlier dataset versions.

- Paper F [9] presents LOTHBROK, an approach to optimizing SPARQL queries over decentralized knowledge graphs. To do this, LOTHBROK proposes a fragmentation technique based on characteristic sets and uses star-shaped query decomposition similar to WISEKG. Furthermore, LOTHBROK proposes an indexing scheme that builds on PPBF indexes and associates the objects in the fragment with the predicates they occur in triples with, called *SPBF indexes*. To optimize a query, LOTHBROK builds a query execution plan in consideration of cardinality estimations based on the SPBF indexes similar to [34, 67, 70], the compatibility of the fragments (whether or not two fragments produce join results for the given query), and the locality of the data (on which nodes the fragments are located). In building the execution plan, LOTHBROK is able to delegate subqueries to nodes in such a way that the overall network usage is minimized. A comprehensive experimental evaluation that compares LOTHBROK to the state-of-the-art systems PIQNIC [5] and COLCHAIN [7] shows a performance increase of up to two orders of magnitude while the network usage is lowered significantly.

The overall goal of this thesis was to take the Semantic Web one step closer to realizing its full potential by solving some of the biggest problems it faces today; availability, scalability, and updatability of the data. Indeed, the papers presented in this thesis provide robust algorithmic and architectural solutions to the problem stated above. Moreover, our work highlights some interesting future work directions as discussed in Section 9.

9 Future Research Directions

As the popularity of the Semantic Web increases, the burden on the data providers to maintain access to the data and keep the data up to date increases as well. In this thesis, we have presented novel frameworks and theoretical models that relieve this burden. However, our work also highlights some interesting and important future research necessary for the Semantic Web to realize its full potential.

As highlighted in [12], decreasing the communication overhead in decentralized systems is only advantageous to a certain degree; it is equally important to balance out the query processing load across the nodes. And while LOTHBROK [9] has taken the first step in this direction by letting nodes

delegate entire subqueries to other nodes in the network, it does not consider the load on each node, and thus risks sending subqueries to already overloaded nodes. Thus, extending the framework presented by Lothbrok with a cost model like the one presented by WiseKG [18] could further improve the overall stability and efficiency of the network under load.

Furthermore, the work presented in this thesis considers two possible data fragmentation techniques (1) predicate-based fragmentation, and (2) fragmentation based on characteristic sets. However, it is still unclear what impact other possible fragmentation and allocation strategies would have on factors such as network usage and query performance. As such, further research is needed to assess multiple different fragmentation and allocation techniques, e.g., based on SHACL/ShEx shapes [73, 74].

While ColChain [7] solves the updatability issue using community-wide consensus that lets users collaborate on keeping the data up to date, it relies on the users to actively vote on whether or not they think an update should be accepted. However, this is not always feasible since in large communities, it could be difficult to get a majority of users to vote on an update, let alone accept it. As such, in the future, applying different consensus protocols, e.g., by letting fragment owners specify a qualified majority or automatically detecting malicious updates, could be important to increase the applicability.

Thus far, we have mainly focused on processing BGP queries. However, the SPARQL language [24] contains further constructs, such as aggregation and analytical queries [31, 49–51]. In the future, it is therefore important to expand the scope of the solutions mentioned in this paper to support such queries as well. Furthermore, expanding the scope of the solutions presented in this thesis to support exploratory or query-by-example approaches [58, 59] could be interesting

Last, our approaches are limited to providing the plain set of answers to the queries; in the future, it could be interesting to expand our frameworks to support provenance information for the data [15, 35, 39], so that the system has information about the origin of the data, and for queries [44], such that the users can obtain explanations on how the query answers were obtained.

References

[1] M. Acosta and M. Vidal, "Networks of linked data eddies: An adaptive web query processing engine for RDF data," in *ISWC 2015*, 2015, pp. 111–127.

[2] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "ANAPSID: an adaptive query processing engine for SPARQL endpoints," in *ISWC 2011*, 2011, pp. 18–34.

[3] M. Acosta, A. Zaveri, E. Simperl, D. Kontokostas, S. Auer, and J. Lehmann, "Crowdsourcing linked data quality assessment," in *ISWC 2013*, 2013, pp. 260–276.

[4] C. Aebeloe, I. Keles, G. Montoya, and K. Hose, "Star pattern fragments: Accessing knowledge graphs through star patterns," *CoRR*, vol. abs/2002.09172, 2020. [Online]. Available: https://arxiv.org/abs/2002.09172

[5] C. Aebeloe, G. Montoya, and K. Hose, "A decentralized architecture for sharing and querying semantic data," in *ESWC 2019*, 2019, pp. 3–18.

[6] ——, "Decentralized indexing over a network of RDF peers," in *ISWC 2019*, 2019, pp. 3–20.

[7] ——, "Colchain: Collaborative linked data networks," in *WWW 2021*. ACM / IW3C2, 2021, pp. 1385–1396.

[8] ——, "A demonstration of colchain: Collaborative knowledge chains," in *ISWC 2021 Posters, Demos and Industry Tracks*, ser. CEUR Workshop Proceedings, vol. 2980, 2021.

[9] ——, "Optimizing sparql queries over decentralized knowledge graphs," in *Unpublished manuscript*, 2022.

[10] C. Aebeloe, G. Montoya, V. Setty, and K. Hose, "Discovering diversified paths in knowledge bases," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 2002–2005, 2018.

[11] C. Aebeloe, V. Setty, G. Montoya, and K. Hose, "Top-k diversification for path queries in knowledge graphs," in *ISWC 2018 Posters & Demonstrations*, ser. CEUR Workshop Proceedings, vol. 2180. CEUR-WS.org, 2018.

[12] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "Dbmss on a modern processor: Where does time go?" in *VLDB 1999*, 1999, pp. 266–277.

[13] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of RDF data management systems," in *ISWC 2014*, 2014, pp. 197–212.

[14] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "CAPER: A cross-application permissioned blockchain," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1385–1398, 2019.

[15] A. B. Andersen, N. Gür, K. Hose, K. A. Jakobsen, and T. B. Pedersen, "Publishing Danish Agricultural Government Data as Semantic Web Data," in *JIST*, vol. 8943, 2014, pp. 178–186.

[16] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?" in *ISWC 2013*, 2013, pp. 277–293.

[17] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives, "Dbpedia: A nucleus for a web of open data," in *ISWC 2007*, 2007, pp. 722–735.

[18] A. Azzam, C. Aebeloe, G. Montoya, I. Keles, A. Polleres, and K. Hose, "Wisekg: Balanced access to web knowledge graphs," in *WWW 2021*. ACM / IW3C2, 2021, pp. 1422–1434.

[19] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, and A. Polleres, "SMART-KG: hybrid shipping for SPARQL querying on the web," in *WWW 2020*. ACM / IW3C2, 2020, pp. 984–994.

[20] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[22] M. Cai and M. R. Frank, "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *WWW*, 2004, pp. 650–657.

[23] A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos, "Semagrow: optimizing federated SPARQL queries," in *SEMANTiCS 2015*. ACM, 2015, pp. 121–128.

[24] W. W. W. Consortium *et al.*, "Sparql 1.1 overview," 2013.

[25] ——, "Rdf 1.1 concepts and abstract syntax," 2014.

[26] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *ICDCS 2002*. IEEE Computer Society, 2002, pp. 23–32.

[27] M. Dumontier, A. Callahan, J. Cruz-Toledo, P. Ansell, V. Emonet, F. Belleau, and A. Droit, "Bio2rdf release 3: A larger, more connected network of linked data for the life sciences," in *ISWC 2014 Posters & Demonstrations Track*, vol. 1272, 2014, pp. 401–404.

[28] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "Blockchaindb - A shared database on blockchains," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1597–1609, 2019.

[29] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[30] L. Galárraga, K. Hose, and R. Schenkel, "Partout: a distributed engine for efficient RDF processing," in *WWW 2014*. ACM, 2014, pp. 267–268.

[31] L. Galárraga, K. A. Jakobsen, K. Hose, and T. B. Pedersen, "Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets," in *ISWC*, 2018, pp. 547–565.

[32] O. Görlitz and S. Staab, "SPLENDID: SPARQL endpoint federation exploiting VOID descriptions," in *COLD 2011*, 2011.

[33] D. Graux, G. Sejdiu, H. Jabeen, J. Lehmann, D. Sui, D. Muhs, and J. Pfeffer, "Profiting from kitties on ethereum: Leveraging blockchain RDF with SANSA," in *ISWC Posters and Demos*, 2018.

[34] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in SPARQL queries," in *EDBT 2014*, 2014, pp. 439–450.

[35] E. R. Hansen, M. Lissandrini, A. Ghose, S. Løkke, C. Thomsen, and K. Hose, "Transparent Integration and Sharing of Life Cycle Sustainability Data with Provenance," in *ISWC*, vol. 12507, 2020, pp. 378–394.

[36] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich, "Data summaries for on-demand queries over linked data," in *WWW 2010*. ACM, 2010, pp. 411–420.

[37] A. Harth, K. Hose, and R. Schenkel, Eds., *Linked Data Management*. Chapman and Hall/CRC, 2014.

[38] O. Hartig and C. B. Aranda, "Bindings-restricted triple pattern fragments," in *OTM 2016 Conferences*, 2016, pp. 762–779.

[39] O. Hartig and et al., "RDF-star and SPARQL-star. W3C Draft Community Group. Report. W3C Community," 2021. [Online]. Available: https://w3c.github.io/rdf-star/cg-spec/2021-12-17.html

[40] O. Hartig, K. Hose, and J. F. Sequeda, "Linked data management," in *Encyclopedia of Big Data Technologies*. Springer, 2019.

[41] L. Heling and M. Acosta, "A framework for federated SPARQL query processing over heterogeneous linked data fragments," *CoRR*, vol. abs/2102.03269, 2021.

[42] L. Heling, M. Acosta, M. Maleshkova, and Y. Sure-Vetter, "Querying large knowledge graphs over triple pattern fragments: An empirical study," in *ISWC 2018*, 2018, pp. 86–102.

[43] J. A. Hendler, J. Holm, C. Musialek, and G. Thomas, "US government linked open data: Semantic.data.gov," *IEEE Intell. Syst.*, vol. 27, no. 3, pp. 25–31, 2012.

[44] D. Hernández, L. Galárraga, and K. Hose, "Computing How-Provenance for SPARQL Queries via Query Rewriting," *Proc. VLDB Endow.*, vol. 14, no. 13, pp. 3389–3401, 2021.

[45] J. V. Herwegen, R. Verborgh, E. Mannens, and R. V. de Walle, "Query execution optimization for clients of triple pattern fragments," in *ESWC 2015*, 2015, pp. 302–318.

[46] A. Hogan, *The Web of Data*. Springer, 2020.

[47] K. Hose, "Knowledge graph (r) evolution and the web of data," in *MEPDaW 2021*, 2021.

[48] K. Hose and R. Schenkel, "Towards benefit-based RDF source selection for SPARQL queries," in *SWIM 2012*. ACM, 2012, p. 2.

[49] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi, "Towards exploratory OLAP over linked open data - A case study," in *BIRTE 2013*, vol. 206, 2014, pp. 114–132.

[50] ——, "Processing aggregate queries in a federation of SPARQL endpoints," in *ESWC 2015s*, vol. 9088, 2015, pp. 269–285.

[51] ——, "Optimizing Aggregate SPARQL Queries Using Materialized RDF Views," in *ISWC*, 2016, pp. 341–359.

[52] A. L. Jakobsen, G. Montoya, and K. Hose, "How diverse are federated query execution plans really?" in *ESWC 2019*, 2019, pp. 105–110.

[53] M. C. Jeffrey and J. G. Steffan, "Understanding bloom filter intersection for lazy address-set disambiguation," in *SPAA 2011*, 2011, pp. 345–354.

[54] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi, "Atlas: Storing, updating and querying RDF(S) data on top of dhts," *J. Web Semant.*, vol. 8, no. 4, pp. 271–277, 2010.

[55] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John, "UniStore: Querying a DHT-based universal storage," in *ICDE 2007*, 2007, pp. 1503–1504.

[56] I. Keles and K. Hose, "Skyline queries over knowledge graphs," in *ISWC 2019*, vol. 11778, 2019, pp. 293–310.

[57] P. Larson, "Dynamic hash tables," *Commun. ACM*, vol. 31, no. 4, pp. 446–457, 1988.

[58] M. Lissandrini, D. Mottin, K. Hose, and T. B. Pedersen, "Knowledge graph exploration systems: are we lost?" in *CIDR 2022*, 2022.

[59] M. Lissandrini, T. B. Pedersen, K. Hose, and D. Mottin, "Knowledge graph exploration: where are we and where are we going?" *SIGWEB Newsl.*, vol. 2020, no. Summer, pp. 4:1–4:8, 2020.

[60] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *VLDB 1986*, 1986, pp. 149–159.

[61] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee, "A demonstration of the solid platform for social web applications," in *WWW 2016*. ACM, 2016, pp. 223–226.

[62] J. P. McCrae, A. Abele, P. Buitelaar, R. Cyganiak, A. Jentzsch, V. Andryushechkin, and J. Debattista, "The linked open data cloud." [Online]. Available: https://lod-cloud.net/

[63] T. Minier, H. Skaf-Molli, and P. Molli, "Sage: Web preemption for public SPARQL query services," in *WWW 2019*, 2019, pp. 1268–1278.

[64] G. Montoya, C. Aebeloe, and K. Hose, "Towards efficient query processing over heterogeneous RDF interfaces," in *DeSemWeb@ISWC 2018*, 2018.

[65] G. Montoya, I. Keles, and K. Hose, "Analysis of the effect of query shapes on performance over LDF interfaces," in *ISWC 2019*, 2019, pp. 51–66.

[66] ——, "Querying linked data: An experimental evaluation of state-of-the-art interfaces," *CoRR*, vol. abs/1912.08010, 2019.

[67] G. Montoya, H. Skaf-Molli, and K. Hose, "The odyssey approach for optimizing federated SPARQL queries," in *ISWC 2017*, pp. 471–489. [Online]. Available: https://doi.org/10.1007/978-3-319-68288-4_28

[68] G. Montoya, M. Vidal, and M. Acosta, "A heuristic-based approach for planning federated SPARQL queries," in *COLD 2012*, 2012.

[69] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[70] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *ICDE 2011*, 2011, pp. 984–994.

[71] O. Papapetrou, W. Siberski, and W. Nejdl, "Cardinality estimation and dynamic length adaptation for bloom filters," *Distributed Parallel Databases*, vol. 28, no. 2-3, pp. 119–156, 2010.

[72] J. Pérez, M. Arenas, and C. Gutiérrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 16:1–16:45, 2009.

[73] K. Rabbani, M. Lissandrini, and K. Hose, "Optimizing SPARQL Queries using Shape Statistics," in *EDBT 2021*, 2021, pp. 505–510.

[74] ——, "SHACL and ShEx in the Wild: A Community Survey on Validating Shapes Generation and Adoption," in *WWW'22 Companion, April 25–29, 2022, Virtual Event, Lyon, France*, 2022.

[75] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, pp. 120–126, 1978.

[76] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo, "LSQ: the linked SPARQL queries dataset," in *ISWC 2015*, 2015, pp. 261–269.

[77] M. Saleem, A. Hasnain, and A. N. Ngomo, "LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation," vol. 48, 2018, pp. 85–125.

[78] M. Saleem, A. Potocki, T. Soru, O. Hartig, and A. N. Ngomo, "CostFed: Cost-based query optimization for SPARQL endpoint federation," in *SEMANTiCS 2018*, ser. Procedia Computer Science, vol. 137. Elsevier, 2018, pp. 163–174.

[79] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: Optimization techniques for federated query processing on linked data," in *ISWC 2011*, 2011, pp. 601–616.

[80] M. Sopek, P. Gradzki, W. Kosowski, D. Kuzinski, R. Trójczak, and R. Trypuz, "Graphchain: A distributed database with explicit semantics and chained RDF graphs," in *WWW Companion*, 2018, pp. 1171–1178.

[81] C. Stadler, J. Lehmann, K. Höffner, and S. Auer, "Linkedgeodata: A core for a web of spatial open data," *Semantic Web*, vol. 3, no. 4, pp. 333–354, 2012.

[82] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over linked data," *World Wide Web*, vol. 14, no. 5-6, pp. 495–544, 2011.

[83] P. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan, and C. B. Aranda, "SPARQLES: monitoring public SPARQL endpoints," *Semantic Web*, vol. 8, no. 6, pp. 1049–1065, 2017.

[84] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple pattern fragments: A low-cost knowledge graph interface for the web," *J. Web Semant.*, vol. 37-38, pp. 184–206, 2016.

[85] D. Vrandecic and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Commun. ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[86] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: a survey," *IJWGS*, vol. 14, no. 4, pp. 352–375, 2018.

[87] I. Zouaghi, A. Mesmoudi, J. Galicia, L. Bellatreche, and T. Aguili, "Query optimization for large scale clustered RDF data," in *DOLAP@EDBT/ICDT 2020*, 2020, pp. 56–65.

Part II

Papers

Paper A

WiseKG: Balanced Access to Web Knowledge Graphs

Amr Azzam, Christian Aebeloe, Gabriela Montoya, Ilkcan Keles, Axel Polleres, Katja Hose

The paper has been published in the
Proceedings of the 30th The Web Conference (WWW 2021), pp. 1422-1434, 2021.
DOI: 10.1145/3442381.3449911

Abstract

SPARQL query services that balance processing between clients and servers become more and more essential to handle the increasing load for open and decentralized knowledge graphs on the Web. To this end, Linked Data Fragments (LDF) have introduced a foundational framework that has sparked research exploring a spectrum of potential Web querying interfaces in between server-side query processing via SPARQL endpoints and client-side query processing of data dumps. Current proposals in between typically suffer from imbalanced load on either the client or the server. In this paper, to the best of our knowledge, we present the first work that combines both client-side and server-side query optimization techniques in a truly dynamic fashion: we introduce WiseKG*, a system that employs 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. Our experiments show that* WiseKG *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.*

Azzam A., Aebeloe C., Montoya G., Keles I., Polleres A., Hose K. (2021) WiseKG: Balanced Access to Web Knowledge Graphs. In: *WWW 2021: Proceedings of the 30th The Web Conference*.
https://doi.org/10.1145/3442381.3449911.
The layout has been revised.

1 Introduction

The Semantic Web has over the past two decades seen a steady increase in the amount of data published as Linked Open Data (LOD), forming a Web of interconnected Knowledge Graphs (KG) [8]. Such KGs are accessible either via public SPARQL endpoints, downloadable data dumps, KG partitions, or dereferenceable URIs. The continued work on Linked Data is fueled by the prospects of such interconnected KGs finally making the vision of an interlinked Web of Data a reality, and at the same time providing scalable query processing to its users. However, as provisioning and maintaining access to KGs is still a huge burden for data publishers [2, 26, 30], low availability of public SPARQL endpoints [5, 29] remains one of the greatest obstacles to this vision.

In order to tackle this bottleneck, various recent proposals emphasize decentralization as a means to lift this burden off the data providers. On one hand, several approaches have focused on the decentralization of the data [2, 3, 9]. While these approaches increase the availability of the data, their query processing capabilities are significantly less efficient than approaches with a powerful centralized server or approaches that ship full data dumps to powerful clients for local processing. On the other hand, several recent studies [1, 6, 12, 19, 20] have focused on the decentralization of the query processing tasks. These approaches divide the processing burden between servers and clients. Even though the servers might be powerful, they will struggle with highly concurrent query loads. Therefore, the clients, which might have free resources, will take some of the query processing tasks for themselves rather than waiting for an overloaded server.

To this end, Triple Pattern Fragments (TPF) [30] reduces the server load significantly by processing joins on the client-side while only processing individual triple patterns on the server. To avoid processing non-selective triple patterns on the server, the client locally processes joins using previously obtained bindings in the request (one binding at a time), potentially leading to smaller intermediate results. Yet, this kind of processing potentially leads to a large number of server requests during query processing, creating a significant overhead on the network traffic. As opposed to just providing triple pattern execution on the server and full join capabilities on the client, two approaches have recently been proposed to optimize SPARQL query processing. Star Pattern Fragments (SPF) [1] exploits server-side evaluation of star-shaped subqueries, while smart-KG [6] exploits client-side evaluation of star-shaped subqueries by retrieving compressed KG partitions from the server. However, the potential benefit of being able to dynamically switch between strategies based on the current server load, caching, etc., remains mostly unexploited by the current state of the art.

Hence, in this paper, we present WiseKG, a novel approach that combines the strengths of the state of the art and further advances them by finding a novel balance between server and client load. Based on the current load on the server, WiseKG decides whether subqueries should be processed on the client or on the server. The underlying cost model considers parameters such as CPU load, estimated network transfer time, and currently available resources at the client to determine where to process a particular part of the query. By applying this 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.

In summary, we make the following contributions:

- We present WiseKG, a novel system that dynamically shifts the query processing load between client and server.
- WiseKG employs a 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.
- Our extensive evaluation 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.

The remaining sections are organized as follows. We cover background in Section 2. Section 3 provides a motivating example. Section 4 gives an overview of WiseKG, followed by a presentation of the server-side cost model. Section 5 details SPARQL query processing on the client- and server-side. Section 6 then presents an empirical evaluation of WiseKG. Last, we conclude the paper and provide an outlook on future work in Section 7.

2 Background

RDF and SPARQL. We assume the readers' familiarity with base technologies such as RDF and SPARQL, from which we borrow standard notation such as RDF Turtle[1] or algebra operators [25]; by $subj(t)$, $pred(t)$, $obj(t)$ we to refer to the components of a single *RDF triple* $t \in G$, such that these components are RDF terms (i.e. URIs/IRIs, blank nodes, and literals). An RDF knowledge graph (KG) G is a set of such triples, where $subj(G)$, $pred(G)$, $obj(G)$ denote subjects, predicates, and objects in G.

RDF KGs can be queried using the query language SPARQL, which relies on matching graph patterns for easy access to RDF stores. The fundamental

[1] http://www.w3.org/TR/2014/REC-turtle-20140225/

graph pattern is a *triple pattern* tp, which is an RDF triple that permits variables from an infinite set V of variables, disjoint with the previously mentioned RDF terms.

A *Basic Graph Pattern (BGP)* is a set of triple patterns $\{tp_1 \dots tp_n\}$ that can be viewed as a conjunctive query; note that while semantically, order is not relevant, we use sequences $(\dots)$ instead of sets $\{\dots\}$ in this paper to indicate execution (left-linear) order in a query plan, i.e., for instance $(tp_1, \dots, tp_n)$ stands for a left-linear query execution plan $(\dots(tp_1 \bowtie tp_2) \bowtie \dots) \bowtie tp_n)$, whereas non-left-linear plans will be denoted by respective explicit parentheses.

For any pattern P, we denote by $var(P)$ its variables. The solutions (or, answers, resp.) of a (query) pattern P over a graph G, denoted $[[P]]_G$, are given as sets Ω of *bindings*, i.e., mappings of the form, $\mu : var(P) \rightarrow R$ to the set R of RDF terms, such that $G \models \mu(P)$, i.e. $\mu(P)$ forms a (sub)graph entailed by G. Two mappings $\mu_1, \mu 2$ are called *compatible*, denoted as $\mu_1 || \mu_2$ if for any $v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$, cf. [25] for details.

A *star pattern* $sp = \{tp_1 ... tp_k\}$ is a BGP such that $subj(tp_i) = subj(tp_j)$ for all $i, j \in \{1, \dots, k\}$, i.e., the subjects of all triple patterns are the same. We refer to k as the *star-size* of sp.

Note that each complex BGP P can be decomposed into a set of star patterns $\mathcal{S}(P)$, called the *(star-)decomposition* of P as follows:

$$\mathcal{S}(P) = \{\{t \in P \mid subj(t) = s\} \mid s \in subj(P)\}$$

Along the above-mentioned notation for query plans as sequences of patterns being interpreted as left-linear query plans, we will analogously write query plans that evaluate patterns per stars as permutations of $\mathcal{S}(P)$, e.g., the query plan shown in Fig. A.1b could be written as (sp_1, sp_2, tp_6), indicating an execution plan at the level of joining star patterns as follows: $((sp1 \bowtie sp2) \bowtie tp)$.

The primary focus of this paper is on evaluating BGPs as the fundamental retrieval functionality of SPARQL. However, more complex patterns, such as *Union*, *Optional*, and *Filters*, are covered by our proposed system, which implements the full SPARQL specification – for a more complete formalization we refer to [25].

2.1 Existing KG Interfaces

In this section, we define query interfaces for KGs following the principles set by Linked Data Fragments (LDF) [30]. 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 with queries or to control the "chunk size" of results through specifying page sizes into which the results should be batched. Note that in line with LDF [30] we also assume G to be blank node-free.

In this paper, we omit details on LDF, such as metadata that is sent along with query results and hypermedia controls. However, we borrow from the original specification [30] and align formal definitions and notations to uniformly present different APIs:

Definition A.1 (adapted from [30])
An LDF API of a KG G accessible at an endpoint URI u[2] is a tuple $f = \langle s, \Phi \rangle$ with

- a selector function $s(G, P, \Omega)$ that defines how a fragment $\Gamma \subseteq G$, or alternatively a set of fragments[3] $\Gamma^* \subseteq 2^G$ is constructed upon calls to the API.
- a paging mechanism $\Phi(n, l, o)$ parameterized by $n, l, o \in \mathbf{N}_0$ denoting maximum page size, limit, and offset.

The selector function s is parameterized by a graph G, pattern P, and a set of bindings Ω[4], where we define two variants, $s(\cdot)$ and $s^*(\cdot)$, which differ essentially in terms of returning either a single graph or one subgraph per solution $\mu \in [[P]]_G$:

$$s(G, P, \Omega) = \{t \in \mu(P) \mid \exists \mu \in [[P]]_G : G \models \mu(P) \wedge (\exists \mu' \in \Omega : \mu' || \mu)\}$$

$$s^*(G, P, \Omega) = \{\mu(P) \mid \exists \mu \in [[P]]_G : G \models \mu(P) \wedge (\exists \mu' \in \Omega : \mu' || \mu)\}$$

As we will see, all LDF APIs discussed in this section can indeed be expressed in terms of one of these two default selector functions.

The general paging mechanism Φ we use in this paper shall enable returning the result in batches, e.g., for LDF use cases where Γ (or, resp., Γ^*) would be very large or retrieving the whole result is not required or possible. Hence, we assume that $\Phi(n, l, o)$ simply defines a mechanism to divide Γ

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

[3]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.

[4]We note that this strict definition of allowed parameters for s is not made in [30], but we will rather use those here to describe the considered APIs uniformly.

into partitions (or *pages*) $\{\Gamma_1, \ldots, \Gamma_k\}$, 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 to request the pages from Γ_o to Γ_{o+l}[5]. We assume l to default to $l = \infty$, o to default to $o = 1$, and finally $n = \infty$ signifying that whole graph Γ should be returned.

In the following, we will explain different approaches on the LDF framework's spectrum, wrt. implications on server availability under high numbers of concurrent clients:

Data Dumps offer the clients simple access to the entire KG. In order to perform a SPARQL query, the clients have to download the whole KG and run a local SPARQL engine themselves. This can be a very beneficial solution for many clients with sufficient resources but puts high processing cost on the client, plus the need for high amounts of data transfers whenever the KG evolves/changes. Data dumps can be characterized in terms of LDF by

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

SPARQL endpoints provide efficient querying on the server side; the query shipped to the server is typically evaluated in an efficient triple store such as Virtuoso, Blazegraph, and Jena, etc., without work for the clients, who receive the ready end result. This 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 $\mu(P)$[6],
- any pattern P 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 $n = |P|$; however, note that subsequent calls of SPARQL queries with consecutive OFFSETs are in general not guaranteed to behave deterministically.

SaGe is – in essence – a SPARQL endpoint with the ability to interrupt

[5] As such l,o should be viewed synonmous SPARQL's LIMIT and OFFSET modifiers.

[6] Deriving μ is straightforward since, given P, μ and $\mu(P)$ 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 [30] – 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.

queries under too much concurrent load on the server side [19]. That is, in principle, we can view SaGe as a variant of SPARQL endpoints that, given a query P whose execution exceeds a timeout τ, suspends it and only returns a partial result $\{\Gamma_1, \ldots, \Gamma_{o-1}\}$, along with additional state information to the client. The client can (with additional hypermedia controls using this state information) deterministically continue exactly at offset o in a subsequent call. Hence, in times of high query load, SaGe uses this strategy to suspend clients to avoid starvation of others. We note that, while the SaGe *server* itself is stateless, i.e., it does not store the intermediate states of the suspended queries, it handles the overall query execution load incl. join processing for BGPs.

Triple Pattern Fragment (TPF) [30] is an interface to enable live SPARQL querying with high availability and scalability by restricting server capabilities to only answer single triple pattern fragments and shifting processing of more complex patterns to the client-side (with the expenses of a substantial increase in the network traffic). In terms of the generic LDF framework, TPF is the most straightforward "incarnation", defined as:

- 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.

Binding-Restricted Triple Pattern Fragments (brTPF) [12] is an extension of TPF that reduces the network load through additionally permitting arbitrary $\Omega \neq \emptyset$. This ensures fewer requests to the server plus faster query processing. However, brTPF still potentially struggles with high numbers of concurrent clients or queries with large intermediate results.

Star Pattern Fragments (SPF) [1] proposes to generalize brTPF from single triples to handling star-shaped subqueries on the *server*. 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 [25]. 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:

- $s_{SPF}(G, P, \Omega) = s^*(G, P, \Omega)$, 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 to paginate over solutions, i.e., retrieving results in chunks of l (iterating over increasing offsets $o := o + l$).

Experiments [1] show that SPF (compared to brTPF) can decrease the number of requests made to the server and intermediate result sizes transferred to the client, maintaining a comparably low network load.

smart-KG [6] (SKG) is another alternative paradigm that combines TPF with the idea to ship graph partitions per star-shaped patterns. To this end, the server holds (compressed and queryable) partitions per common predicate families of G, defined as follows:

Definition A.2 (Predicate family)
We define a *predicate family* $F(s)$ wrt. to KG G as the set of predicates associated with subject s:

$$F(s) = \{p \mid \exists o \in obj(G) : (s, p, o) \in G\} \tag{A.1}$$

We denote the set of families of a graph G as $F(G)$ or F for simplicity whereby $F(G) = \{F(x) | x \in subj(G)\}$.

Predicate families, also known as *characteristic sets*, were introduced in [23] as an RDF query cardinality estimation method while SKG uses families as a basis for inducing a graph partitioning of G, with one partition G_f per $f \in F(G)$ [6].

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. Evaluations [6] show that this approach is highly competitive for many concurrent clients due to its low server and (due to partition compression also) network footprint. 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.

2.2 RDF HDT Compression

It is worthwhile to also explain the HDT [10] binary compression format for RDF datasets that is used "under the hood" in all of the previously mentioned interfaces, namely (br)TPF, SaGe, SKG, and SPF, as well as in our novel approach presented in this paper. HDT offers efficient search and retrieval over the compressed RDF graphs without the need for decompression and offering query relevant statistics directly in its metadata. The main compression idea relies on ordering triples by *SPO*, grouping repetitive RDF terms. An HDT file could be viewed as a compressed, directly queryable *SPO*-ordered index. In addition, HDT provides a compressed binary utility index built upon loading time covering *OPS PSO* to achieve a high performance for resolving any SPARQL triple pattern. TPF and SPF rely on an HDT of the whole graph G to evaluate triple and star patterns on the server with a low computation footprint, whereas SKG profits from the compression also lowering the network footprint when shipping family partitions G_f.

3 Motivating Example

All thus far described KG APIs alone suffer from an imbalanced load on either client-side (dumps, TPF, SKG) or server-side (SPARQL endpoints, SaGe, SPF). In this paper, 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.

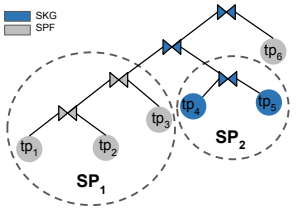
In particular, the factors that our cost model considers are server load, client computing resources, and the number/size of intermediate results to be transferred over the network (in combination with available bandwidth), since several sources [1, 6, 13, 21, 22] identified these as important dimensions when accessing KGs.

To elaborate, let us consider query Q given in Figure A.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 of a 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 $(sp_1^{SPF}, sp_2^{SKG}, tp_6^{SPF})$

Fig. A.1: Example of processing a SPARQL query with WiseKG

4 WiseKG

In the spirit of the example presented in Section 3, 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.

4.1 Overview

WiseKG employs a dynamic cost model to determine an *annotated query plan*: in order to denote query execution plans (cf. Section 2) 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 case of the example in Figure A.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[7].

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 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 $s_{SKG}(G, P, \emptyset)$[8],
- 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, in this paper 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 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 $\tau = \tau_C + \iota$, where τ_C corresponds to the current time, and ι is a

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

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

fixed time quantum per query[9]. 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), we first present the server cost model, which is used to make this latter choice.

4.2 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 [18] from the field of distributed databases [18, 31]. 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 client and server to process a star subquery: estimated number of CPU instructions ($\#CPU$), estimated number of I/O operations ($\#IO$), as well as two communication cost components – 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

$$cost(sp) = \underbrace{W_{CPU} \times (\#CPU)}_{\text{Processing}} + \underbrace{W_{MSG} \times (\#M)}_{\text{Messaging}} + \underbrace{W_{BYT} \times (\#BYT)}_{\text{Data transfer}} + \underbrace{W_{IO} \times (\#IO)}_{\text{I/O}} \tag{A.2}$$

where the weights W_{CPU}, W_{MSG}, W_{BYT}, and W_{IO} help estimating 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:

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[10]:

[9] Somewhat similar to/inspired by SAGE's [19] query suspension timeouts.

[10] We estimate this current CPU load as the average percentage of CPU_{usage} in the previous

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

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 [19], we assume as 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 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 [6]. 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 [10]). 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 6.1), 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 A.3 and Definition A.4. For a query BGP P, these costs are estimated for each star pattern $sp \in \mathcal{S}(P)$.

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 A.3 (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 using SKG is estimated as follows:

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

Definition A.4 (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:

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

Definitions A.3 and A.4 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 [23].
- $size(\cdot)$ 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[11].
- 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. [10]; 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 $W_{CPU_{client}}$.
- 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.

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

5 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.

5.1 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 [6] and [1], 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. 8.

Algorithm 8 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: $S \leftarrow \mathcal{S}(P)$
3: $\Pi_p \leftarrow ()$
4: **while** $S \neq \emptyset$ **do**
5: **for** $sp \in S$ **do**
6: $cnt_{sp} \leftarrow card(sp, \Pi_p)$
7: **if** $cnt_{sp} = 0$ **then return** $()$
8: $sp_i \leftarrow sp$ where $sp \in S$ and $cnt_{sp} \leq cnt_{sp'}$ for all $sp' \in S$
9: **if** $cost_{SPF}(sp_i, \Pi_P) \leq cost_{SKG}(sp_i, \Pi_P)$ **then**
10: $\Pi_P \leftarrow append(\Pi_P, (sp_i^{SPF}))$
11: **else**
12: $\Pi_P \leftarrow append(\Pi_P, (sp_i^{SKG}))$
13: $S \leftarrow S \setminus \{sp_i\}$
14: $\tau \leftarrow \tau_{\mathcal{C}} + \iota$
15: **return** (Π_P, τ)

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 5-7) and add it to the plan; when we find a query with an empty result (e.g. in case no matching family partition exists [6]), we can stop since the final result will then also be empty. The star pattern with the lowest cardinality estimation is selected first (line 8), 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 4.2, each subquery is annotated with the resp. control for evaluating the star pattern on the server, i.e., *SPF* (line 10) or the client *SKG* (line 12). 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 15) after computing the expiry timestamp (line 14).

For the query Q shown in Figure A.1a, this algorithm could compute the execution plan in the join order visualized in Figure A.1b (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. [6, Section 4.1]); in such cases the concrete implementation of Alg. 8 defaults to SPF, i.e., server-side evaluation of the resp. star pattern, independent of the cost.

5.2 Client-Side Query Processing

Processing queries on a WiseKG client relies on an approach similar to the one presented in [1], which we adapt herein to accommodate for client-side processing of HDT 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.,[12] 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.
   (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 $c = SKG$ involves client-side evaluation of the star-shaped pat-

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

tern 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. 9.

Algorithm 9 Processing a Query Execution Plan

Input: $\Pi = (sp_1^{c_1}, \ldots, sp_n^{c_n})$ // 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** $\tau < \tau_C$ **then**
3: $(\Pi, \tau) \leftarrow Plan(BGP(\Pi))$
4: **if** $\Pi = sp^c$ **then**
5: $\Omega \leftarrow eval_c(sp, \Omega')$
6: **else**
7: $\Omega \leftarrow evalPlan((sp_1^{c_1}, \ldots, sp_{n-1}^{c_{n-1}}), \tau, \Omega')$
8: $\Omega \leftarrow evalPlan(sp_n^{c_n}, \tau, \Omega)$
9: **return** Ω

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)[13]. The way this is currently done can be understood as follows: assuming the originally requested plan is $(sp_1^{c_1} \ldots sp_i^{c_i} \ldots sp_n^{c_n})$ and the client reaches τ at step i. Then the client will restart calling $Plan(\{sp_i, \ldots sp_n\})$ receiving a new plan $\Pi_{\{sp_i, \ldots sp_n\}}$ 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. 9, 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).

6 Experimental Evaluation

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

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

6.1 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[14] extending the TPF implementations[15] 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 [6] 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[16] 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) [4] to test the scalability of our approach: 10M, 100M, and 1B triples. In addition to these, we also use the real-world dataset DBpedia [16] (v.2015A). The characteristics of the evaluated RDF graphs are described in Table A.1.

Table A.1: Characteristics of the used datasets

Dataset	#triples	#subjects	#predicates	#objects	#families
`watdiv10M`	10,916,457	521,585	86	1,005,832	21,210
`watdiv100M`	108,997,714	5,212,385	86	9,753,266	37,392
`watdiv1B`	1,092,155,948	52,120,385	86	92,220,397	52,885
`DBpedia`	837,257,959	113,986,155	60,264	221,623,898	29,965

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[17]. Each client has a set

[14] https://github.com/WiseKG/WiseKG-Java
[15] https://github.com/LinkedDataFragments/Server.java
[16] https://github.com/oshi/oshi
[17] https://dsg.uwaterloo.ca/watdiv/basic-testing.shtml

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 [4]. 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 [27]; plus, we included 12 queries used to evaluate smart-KG [6].

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 disregarding the cost model defined in Section 4.2, we implemented a version of WiseKG named WiseKG$_{heuristic}$ that relies on more straightforward heuristics. Initially, WiseKG$_{heuristic}$ executes all star subqueries on the server side up to a predefined CPU usage threshold σ. When the threshold is reached, WiseKG$_{heuristic}$ produces an execution plan exclusively based on shipping family partitions. In addition, we evaluate WiseKG$_{cost}$, our main contribution, which is a version of WiseKG that relies on the cost model described in Section 4.2. Note that we use the recommended versions of both server and client for all the evaluated systems including Star Pattern Fragment (SPF) [1], smart-KG [6], SaGe [19], and Triple Pattern Fragments (TPF) [30].

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 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.

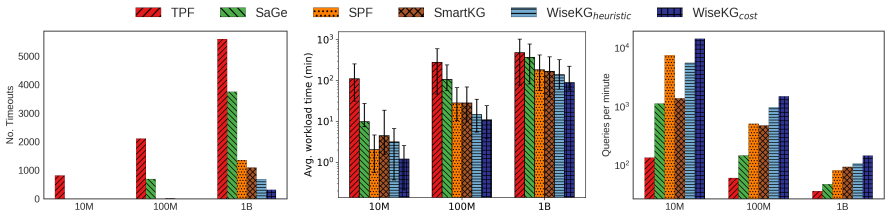
Evaluation metrics.

- **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 time it 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 [1, 6, 19], 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 [1, 30]) and the maximum number of bindings attached to a request for SPF and WiseKG was set to $|\Omega| = 30$ as it was in [1]. 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.[18]

6.2 Experimental Results

Due to space restrictions, this section focuses on the most important results. All results, incl. additional experiments, details on the implementation and configurations used in the experiments (datasets and queries) are available online[19].



(a) Number of Timeouts **(b)** Avg. Workload Time **(c)** Queries per minute

Fig. A.2: Number of timeouts, average workload time, and throughput for 128 clients over `watdiv10M`, `watdiv100M`, and `watdiv1B` on `watdiv-sts`

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 A.2, 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 WiseKG$_{cost}$ (just WiseKG hereafter) has even surpassed WiseKG$_{heuristic}$.

Figure A.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

[18] 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.

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

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 A.2b shows that 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 A.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. To provide a comprehensive evaluation, we also include TPF and SaGe in our experiments. As shown in Figure A.2, our experiments confirm a previous study [6] 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 [1, 6, 19, 30] have already shown that SPARQL endpoints are not able to scale well with an increasing number of clients.

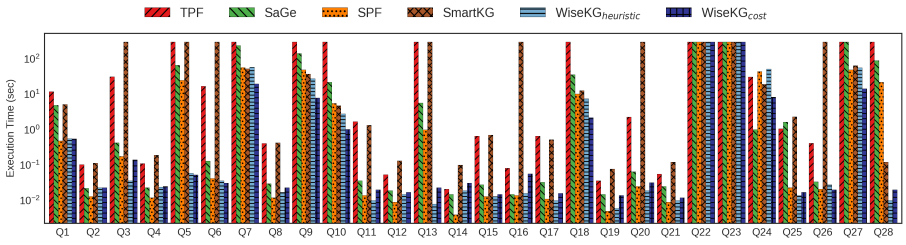
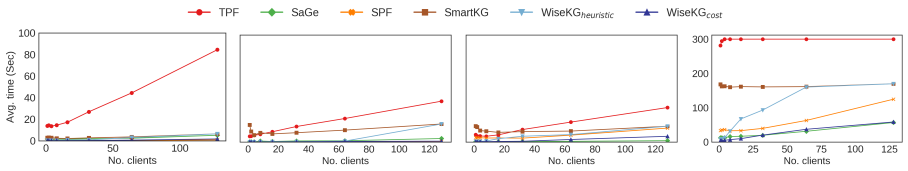


Fig. A.3: Execution time (in seconds) for 28 diverse queries over the `dbpedia` dataset.

We compare the performance of WiseKG to state-of-the-art interfaces considering real-world queries on DBpedia. Figure A.3 presents the execution times of these 28 queries for all systems. The results confirm that WiseKG sig-

nificantly outperforms the compared systems for the real-world queries. Figure A.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, WiseKG$_{heuristic}$ is faster than WiseKG for the queries with execution time less than 0.1 seconds. This is because WiseKG has the overhead of computing the best query plan.



(a) L1-L5 **(b)** S1-S7 **(c)** F1-F5 **(d)** C1-C3

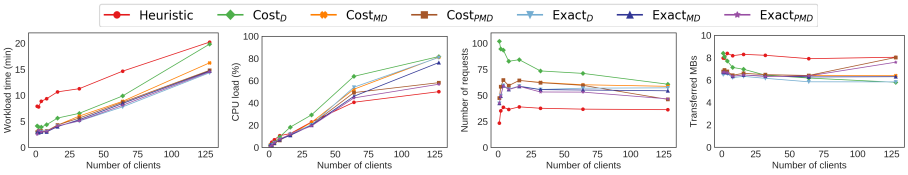
Fig. A.4: Avg. execution time per client over `watdiv100M` for the `watdiv-btt` workload.

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 queries of 4 different shapes including linear (L), star (S), snowflake (F), and complex (C) queries. These queries are part of the `watdiv-btt` workload and executed against `watdiv100M`. The `watdiv-btt` queries were executed in a different (random) order on each client and the results were recorded as the overall execution time per query shape averaged across all clients. Figure A.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. A.4b, SPF provides an 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. A.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. A.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, 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.



(a) Avg. Workload Time **(b)** CPU Usage **(c)** Avg. requests per query **(d)** Data transfer/query

Fig. A.5: Impact of the cost model components on the performance and resources consumption over `watdiv100M`

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 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 WiseKG$_{heuristic}$ as baseline. Figure A.5a shows that for

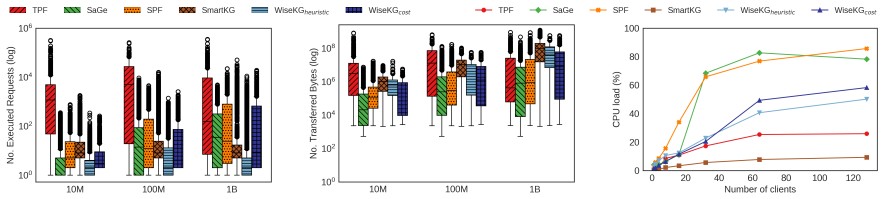
the configuration with 128 clients $Cost_{PMD}$ improves the average workload completion time (14 min) compared to $Cost_D$ and $Cost_{MD}$ (19min and 16min, respectively). In addition, Figures A.5b and A.5c show that $Cost_{PMD}$ requires on average less CPU usage and number of requests than $Cost_D$ 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 WiseKG$_{heuristic}$, and since WiseKG$_{heuristic}$ is faster than all the state-of-the-art systems (Figure A.2), so are all the configurations.

Moreover, to evaluate the impact of using *characteristic set* [23] 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 A.5b, A.5c, and A.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 A.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 a 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 [17, 24].

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 client and server (NTB). Figures A.6a and A.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 the data transfer leading to a substantial increase in network



(a) No. 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).

(c) Avg. Server CPU Usage (in percentage) for increasing numbers of clients over `watdiv1B`.

Fig. A.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

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. Figure A.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 A.5c; second, WiseKG, in contrast to smart-KG, ships bindings along with the triple pattern requests (as presented in brTPF [12] 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 A.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 A.6c 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 A.6c, 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 A.6c 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 A.2a).

7 Conclusions and Future Work

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 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, and 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 high 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.

In our future work, we plan to evaluate in more detail the influence of different hardware setups and mixes of clients with differing computational 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 [14, 15]. Moreover, we plan to extend our implementation, which is currently implemented as a standalone setup, into a framework that flexibly allows to integrate different LDF APIs [20] and also other cost models. The recently introduced Comunica [7, 28] platform could serve as a starting point for integration. 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 [11] could address this current limitation.

References

[1] C. Aebeloe, I. Keles, G. Montoya, and K. Hose, "Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns," *CoRR*, vol. abs/2002.09172, 2020.

[2] C. Aebeloe, G. Montoya, and K. Hose, "A decentralized architecture for sharing and querying semantic data," in *ESWC 2019*, 2019, pp. 3–18.

[3] ——, "Decentralized indexing over a network of RDF peers," in *ISWC 2019*, 2019, pp. 3–20.

[4] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of RDF data management systems," in *ISWC 2014*, 2014, pp. 197–212.

[5] C. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche, "SPARQL web-querying infrastructure: Ready for action?" in *ISWC 2013*, 2013, pp. 277–293.

[6] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, and A. Polleres, "SMART-KG: hybrid shipping for SPARQL querying on the web," in *WWW 2020*, 2020, pp. 984–994.

[7] A. Azzam, R. Taelman, and A. Polleres, "Towards cost-model-based query execution over hybrid linked data fragments interfaces," in *ESWC 2020*, 2020, pp. 9–12.

[8] P. A. Bonatti, S. Decker, A. Polleres, and V. Presutti, "Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web (Dagstuhl Seminar 18371)," *Dagstuhl*, vol. 8, no. 9, pp. 29–111, 2019.

[9] M. Cai and M. R. Frank, "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *WWW 2004*, 2004, pp. 650–657.

[10] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[11] J. D. Fernández, M. A. Martínez-Prieto, A. Polleres, and J. Reindorf, "HDTQ: managing RDF datasets in compressed space," in *ESWC 2018*, 2018, pp. 191–208.

[12] O. Hartig and C. B. Aranda, "Bindings-restricted triple pattern fragments," in *ODBASE 2016*, 2016, pp. 762–779.

[13] L. Heling and M. Acosta, "Cost- and robustness-based query optimization for linked data fragments," in *ISWC 2020*, 2020, pp. 238–257.

[14] D. Ibragimov, K. Hose, T. Pedersen, and E. Zimányi, "Optimizing Aggregate SPARQL Queries Using Materialized RDF Views," in *ISWC 2016*, 2016, pp. 341–359.

[15] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi, "Processing Aggregate Queries in a Federation of SPARQL Endpoints," in *ESWC 2015*, 2015, pp. 269–285.

[16] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. Mendes, S. Hellmann, M. Morsey, P. Kleef, S. Auer, and C. Bizer, "DBpedia - A large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[17] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *PVLDB*, vol. 9, no. 3, pp. 204–215, 2015.

[18] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *VLDB 1986*, 1986, pp. 149–159.

[19] T. Minier, H. Skaf-Molli, and P. Molli, "Sage: Web preemption for public SPARQL query services," in *WWW 2019*, 2019, pp. 1268–1278.

[20] G. Montoya, C. Aebeloe, and K. Hose, "Towards Efficient Query Processing over Heterogeneous RDF Interfaces," in *DeSemWeb@ISWC 2018*, 2018.

[21] G. Montoya, I. Keles, and K. Hose, "Analysis of the effect of query shapes on performance over LDF interfaces," in *QuWeDa@ISWC 2019*, 2019, pp. 51–66.

[22] G. Montoya, M. Vidal, Ó. Corcho, E. Ruckhaus, and C. B. Aranda, "Benchmarking Federated SPARQL Query Engines: Are Existing Testbeds Enough?" in *ISWC 2012*, 2012, pp. 313–324.

[23] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *ICDE 2011*, 2011, pp. 984–994.

[24] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W.-S. Han, "G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching," in *SIGMOD 2020*, 2020, pp. 1099–1114.

[25] J. Pérez, M. Arenas, and C. Gutiérrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 16:1–16:45, 2009.

[26] A. Polleres, M. R. Kamdar, J. D. Fernández, T. Tudorache, and M. A. Musen, "A more decentralized vision for linked data," in *DeSemWeb@ISWC 2018*, 2018.

[27] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo, "LSQ: the linked SPARQL queries dataset," in *ISWC 2015*, 2015, pp. 261–269.

[28] R. Taelman, J. V. Herwegen, M. V. Sande, and R. Verborgh, "Comunica: A modular SPARQL query engine for the web," in *ISWC 2018*, 2018, pp. 239–255.

[29] P. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan, and C. Aranda, "SPARQLES: monitoring public SPARQL endpoints," *Semantic Web*, vol. 8, no. 6, pp. 1049–1065, 2017.

[30] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: A low-cost knowledge graph interface for the Web," *J. Web Semant.*, vol. 37-38, pp. 184–206, 2016.

[31] I. Zouaghi, A. Mesmoudi, J. Galicia, L. Bellatreche, and T. Aguili, "Query optimization for large scale clustered RDF data," in *DOLAP@EDBT/ICDT 2020*, 2020, pp. 56–65.

Paper B

A Decentralized Architecture for Sharing and Querying Semantic Data

Christian Aebeloe, Gabriela Montoya, Katja Hose

The paper has been published in the
Proceedings of the 16th Extended Semantic Web Conference (ESWC 2019),
pp. 3-18, 2019. DOI: 10.1007/978-3-030-21348-0_1

Abstract

*Although the Semantic Web in principle provides access to a vast Web of interlinked data, the full potential remains mostly unexploited. One of the main reasons for this is the fact that the architecture of the current Web of Data relies on a set of servers providing access to the data. These servers represent bottlenecks and single points of failure that result in instability and unavailability of data at certain points in time. In this paper, we therefore propose a decentralized architecture (*Piqnic*) for sharing and querying semantic data. By combining both client and server functionality at each participating node, and introducing replication,* Piqnic *avoids bottlenecks and keeps datasets available and queryable although the original source might not be available. Our experimental results, using a standard benchmark of real datasets, show that* Piqnic *can serve as an architecture for sharing and querying semantic data, even in the presence of node failures.*

Aebeloe C., Montoya G., Hose K. (2019) A Decentralized Architecture for Sharing and Querying Semantic Data. In: Hitzler P. et al. (eds) The Semantic Web. ESWC 2019. Lecture Notes in Computer Science, vol 11503. Springer, Cham.
https://doi.org/10.1007/978-3-030-21348-0_1
The layout has been revised.

1 Introduction

More and more datasets are being published in RDF format. These datasets cover a broad range of topics, such as geography, cross-domain knowledge, government, life sciences, etc. Access to these datasets is offered in different ways, e.g., they can be downloaded as data dumps, they can be queried via SPARQL endpoints, or they can be "browsed" via dereferencing URIs.

Once published, however, we are often in a situation where the datasets, or rather the interfaces to access them, are not available when needed. In fact, studies found that over half of the public SPARQL endpoints have less than 95% availability [2]. The reason often simply is that maintaining these interfaces requires considerable resources from the data providers. In practice, this means that the data necessary to answer a certain query might not be available at a specific time so that the answer might be incomplete – or in general, the same query might have different answers at different points in time.

Hence, despite the great potential of the Semantic Web, accessing RDF datasets today entirely relies on the services offered by the data providers, e.g., web interfaces with downloadable datasets, SPARQL endpoints, or dereferenceable URIs. Especially SPARQL endpoints often require huge amounts of resources for query processing, which further increases the burden on the data providers [9, 22]. Despite recent efforts that proposed to implement monetary incentives to solve this problem [8], we argue that we can achieve availability by applying decentralization instead of relying on the availability of single servers and their functionality. This not only better reflects the nature of the World Wide Web but also avoids dependencies and single points of failure.

In this paper, we therefore propose PIQNIC (a P2p clIent for Query processiNg over semantIC data). PIQNIC introduces decentralization as a key concept by building on the Peer-to-Peer (P2P) paradigm and replication. PIQNIC functions as a P2P network of homogeneous nodes that can be queried by any node in the network. By combining both client and server functionality at each peer and introducing replicas, we avoid single points of failure as (sub)queries can be processed by multiple alternative peers and the data is still available even though the original source is not. In doing so, PIQNIC offers a solution to one of the main problems that the current Semantic Web is suffering from: availability of datasets [2]. It is therefore not our main goal to enable load sharing among nodes to enhance query processing performance and outperform existing systems. In summary, this paper makes the following contributions:

- A P2P-based architecture for publishing and querying RDF data (PIQNIC)

- A customizable scheme for replicating and fragmenting datasets
- Query processing strategies in PIQNIC networks with replicated and fragmented data
- An extensive evaluation of the proposed approaches

This paper is structured as follows. While Section 2 discusses related work, Section 3 presents the PIQNIC framework and its main concepts. Section 4 then describes how to process queries in PIQNIC. Section 5 then presents the results of our evaluation and Section 6 concludes the paper with a summary and an outlook to future work.

2 Related Work

Recent developments in privacy and personal data on the social Web has inspired interesting new applications and use cases. The Solid project [14], for instance, uses a decentralized architecture and Semantic Web technologies to enable personal online datastores (pod) to be stored separately from applications. In fact, users decide themselves where a pod is hosted, giving them control over their data. While the idea of storing Linked Data in multiple locations is central to our work, Solid focuses on privacy protection of personal data whereas we focus on the availability of open datasets.

Federated query processing over SPARQL endpoints, e.g., [19], is a widely used approach to query over distributed Linked Open Data. To lower the computational load at the servers hosting the SPARQL endpoints, recent proposals, such as Triple Pattern Fragments (TPF) [22] and Bindings-Restricted Triple Pattern Fragments (brTPF) [9], propose to shift part of the load to the client issuing the query [18]. This, in turn, increases the availability of the servers. Nevertheless, TPF/brTPF servers still represent a single point of failure; if the server is not available, the hosted datasets are not available either, which is the problem we are targeting in this paper.

To further share the computational load in a TPF setting, processing SPARQL queries in networks of browsers has been proposed [6, 7, 17]. The key principle is to share the computational load among a set of clients based on the functionality offered by their browsers and caching of recently used datasets using a collaborative caching system based on overlay networks [5]. However, browsers are relatively unstable nodes, with very limited processing power and storage capacity, which naturally limits the general applicability. In contrast, we aim at a relatively stable network with more powerful nodes and datasets split into smaller fragments that are replicated.

Replication of triple pattern fragments has been considered in [16], where fragments are replicated at multiple servers to allow for balancing the server loads and providing fault tolerance. While [16] considers a fixed set of servers

that provide access to a fixed set of replicated fragments, and clients that are aware of the allocation of fragments to servers, PIQNIC has a fault-tolerant P2P-based architecture where clients also serve replicated fragments to other clients, which naturally allows for handling dynamic behavior of the clients.

P2P systems in general vary in their level of decentralization. Structured P2P systems organize their peers in an overlay network using, for instance, Distributed Hash Tables (DHTs) to decide were to store and find particular data items. Some of these systems were proposed to support RDF data [3, 11, 12]. The key principle of these systems is that the connections between peers, i.e., the layout of the network, and the data placement is imposed on the participating peers, restricting their autonomy. As a consequence, such systems are vulnerable to situations where many peers leave and join the network as this might require major reorganizations of structure and data placement in the network.

Unstructured P2P systems, on the other hand, retain a high degree of peer autonomy, i.e., there is no globally enforced network layout or data placement. The basic way of processing queries in such networks is flooding, i.e, a request is flooded through the network along the connections between neighboring peers until an answer has been found. These systems are therefore more reliable with respect to dynamic behavior, i.e., nodes joining and leaving the network. The prospects of unstructured P2P techniques as a decentralized architecture for Linked Data have also been recognized in recent vision papers [15, 20]. While these papers provide interesting insights in the benefits of decentralization and replication, we propose a concrete system and implementation for query processing over an unstructured network of P2P nodes.

3 PIQNIC

PIQNIC builds upon basic principles of P2P systems; a client software is running at each participating peer that (i) provides access to a network of clients without central authority and (ii) offers access to datasets stored locally at other nodes in the network. To minimize local space consumption at a node, we use HDT [4] files. As common in P2P networks, nodes do not have global knowledge of all the peers in the network and their connections. Hence, PIQNIC nodes always maintain a partial view of the entire network. This partial view consists of (i) nodes with related data, i.e. data that uses common URI/IRIs, to ensure that queries over multiple datasets can be completed efficiently and (ii) random neighbors to ensure connectivity of the entire network.

Before going into details on query processing (Section 4), this section first introduces the notion of datasets and data fragments (Section 3.1). After-

wards, Section 3.2 outlines PIQNIC's network architecture. Last, Section 3.3 describes the dynamic behavior of PIQNIC nodes, maintaining a partial view over the network, and data replication.

3.1 Data Fragmentation

Since RDF datasets can be quite large (e.g., YAGO3 [13] with over 100 million triples), replicating entire RDF datasets at another node might not always be possible or useful. Hence, inspired by the TPF style of accessing data, we propose a customizable approach for fragmenting large datasets.

Consider the infinite and disjoint sets U (the set of all URIs/IRIs), B (the set of all blank nodes), L (the set of all literals), and V (the set of all variables). An RDF *triple* t is a triple s.t. $t \in (U \cup B) \times U \times (U \cup B \cup L)$, and a *triple pattern* tp is a triple s.t. $tp \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$. A knowledge graph $\mathcal{G}$ is a finite set of RDF triples.

Definition B.1 (Fragment)
Let $\mathcal{G}_N$ be a knowledge graph that includes all RDF triples in a PIQNIC network. A fragment f is a 4-tuple $f = \langle T, N, u, i \rangle$ with the following elements:

- T is a finite set of RDF triples, and $T \subseteq \mathcal{G}_N$,
- N is a set of PIQNIC nodes storing the fragment,
- u is a URI/IRI that identifies the fragment, and
- i is an identification function that determines whether the fragment contains triples matching a given triple pattern.

Identification functions are mainly used during query processing to determine whether or not a triple pattern should be evaluated over a fragment.

Following the principle that a data provider uploads a dataset using a local PIQNIC node, we say that datasets are "owned" by a specific node. The owner node manages the allocation of replicas to other nodes in the network (details in Section 3.3). We then define a dataset as a set of fragments:

Definition B.2 (Dataset)
A dataset D is a triple $D = \langle F, u, o \rangle$ with the following elements:

- F is a set of fragments,
- u is a URI/IRI that identifies the dataset, and
- o is an identifier of the "owner" node, i.e., the node that uploaded F to the network.

A fragmentation function $\mathcal{F}$ is then defined as follows.

Definition B.3 (Fragmentation function)
A fragmentation function $\mathcal{F}$ is a function that, when applied to a knowledge graph $\mathcal{G}$, creates a set of fragments $F = \mathcal{F}(\mathcal{G})$, i.e., $\mathcal{F}(\mathcal{G}) : \mathcal{G} \mapsto 2^{\mathcal{G}}$.

Concrete fragmentation functions can result in different levels of granularity. For example, a fragmentation function $\mathcal{F}_C$ that results in $\mathcal{F}_C(\mathcal{G}) = \{\mathcal{G}\}$ is a very coarse-granular fragmentation function that does not split up the original knowledge graph $\mathcal{G}$. On the other hand, a fragmentation function $\mathcal{F}_F$ that results in $\mathcal{F}_F(\mathcal{G}) = \{\{t\} \mid t \in \mathcal{G}\}$, creates a separate fragment for each individual triple and is very fine-granular. PIQNIC uses a predicate-based fragmentation function $\mathcal{F}_P$ as defined in Definition B.4.

Definition B.4 (Predicate-based fragmentation function)
Let p_t denote the predicate of a triple t. A predicate-based fragmentation function $\mathcal{F}_P(\mathcal{G}) = \{F_p \mid \exists t \in \mathcal{G} : p_t = p \wedge (\forall t' \in \mathcal{G})[p_{t'} = p] : t' \in F_p\}$ defines one fragment for each unique predicate in the knowledge graph $\mathcal{G}$.

Naturally, more complex fragmentation functions can be defined. However, in its current implementation PIQNIC uses $\mathcal{F}_P$ because it has a straightforward implementation and is guaranteed to generate pairwise disjoint fragments as each triple has exactly one predicate, i.e., for any two fragments $f_i, f_j \in \mathcal{F}_P(\mathcal{G})$ it holds that $f_i \cap f_j = \emptyset$.

Example B.1 (Fragmentation)
Consider example knowledge graph $\mathcal{G}_E$ in Table B.1a. Applying $\mathcal{F}_P$ to $\mathcal{G}_E$ results in the set of fragments f_1, f_2, f_3 f_4, and f_5 shown in Table B.1b; one fragment for each unique predicate p_1, p_2, p_3, p_4, and p_5.

Table B.1: Applying $\mathcal{F}_P$ to a knowledge graph $\mathcal{G}_E$

(a) Knowledge graph $\mathcal{G}_E$

Knowledge graph $\mathcal{G}_E$		
⟨a p_1 b⟩	⟨a p_2 c⟩	⟨a p_3 d⟩
⟨b p_2 e⟩	⟨b p_1 d⟩	⟨b p_3 d⟩
⟨c p_3 d⟩	⟨d p_1 c⟩	⟨c p_2 a⟩
⟨f p_4 d⟩	⟨d p_4 f⟩	⟨e p_5 g⟩

(b) $\mathcal{F}_P(\mathcal{G}_E)$

$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{f_4}$	$\mathbf{f_5}$
⟨a p_1 b⟩	⟨a p_2 c⟩	⟨a p_3 d⟩	⟨f p_4 d⟩	⟨e p_5 g⟩
⟨b p_1 d⟩	⟨b p_2 e⟩	⟨b p_3 d⟩	⟨d p_4 f⟩	
⟨d p_1 c⟩	⟨c p_2 a⟩	⟨c p_3 d⟩		

3.2 Network Architecture

A PIQNIC network consists of a set of interconnected nodes, each maintaining a local data/triple store to manage a set of fragments. A node is defined as follows.

Definition B.5 (Node)
A node n is a triple $n = \langle \Gamma, \Delta, N \rangle$ where

- Γ is the set of fragments located on the node,
- Δ is a set of datasets owned by the node, and
- N is a set of so-called neighbor nodes in the network.

Each node n maintains a set $n.N$ of neighbor nodes representing a partial view over the network. In order to ensure that (i) related data is close in the network to increase the completeness of query answers, and (ii) all data and nodes can be reached (connectivity of the network), $n.N$ contains nodes with related fragments as well as random nodes in the network.

To account for changes in the network, PIQNIC uses periodic shuffles [23] between pairs of nodes. A node n selects a random node n' in $n.N$, which it sends a subset of its neighbors, removing them from its own partial view. This subset consists of the least related neighbors based on the "joinability" of the nodes' fragments.

Definition B.6 (Fragment Joinability)
Let s_t and o_t be the subject and object of triple t, $\mathcal{G}_N$ the knowledge graph containing all RDF triples in a network, and $f_1, f_2 \in \mathcal{F}_P(\mathcal{G}_N)$. f_1 and f_2 are said to be "joinable", denoted $f_1 \perp\!\!\!\perp f_2$, iff for at least one triple $t_1 \in f_1$, there exists a triple $t_2 \in f_2$, s.t. $\{s_{t_1}, o_{t_1}\} \cap \{s_{t_2}, o_{t_2}\} \neq \emptyset$.

We observe that the binary relation $\perp\!\!\!\perp$ is symmetric and reflexive. It is symmetric since if t_1 has a subject or object in common with t_2, t_2 has the same subject or object in common with t_1. It is reflexive since any triple t has its own subjects and objects in common with itself.

Fragment joinability only considers *if* two fragments are joinable, and does not consider the rate of overlap between them. This is to avoid favoring large fragments where the absolute number of joint subjects and objects is likely to be higher than for small fragments because of the higher number of triples. The relative number of overlapping subjects and objects is not a good alternative either as fragments with a small overlap might still be important to achieve complete query results.

Based on Definition B.6, we can now define a relatedness metric to rank a node's neighbors. We consider only non-identical joinable fragments. Hence, given a node n the goal is to select the k least related nodes R, where $R \subseteq n.N$ s.t. we minimize the objective function in Equation B.1.

$$Rel(n) = \underset{R \subseteq n.N}{\arg\min} \sum_{n_i \in R} \frac{|Join(n, n_i)|}{|n.\Gamma|} \qquad s.t.\ |R| = k \tag{B.1}$$

where $Join(n, n_i)$, as defined in Equation B.2, is the set of fragments in n that are joinable with one of node n_i's fragments that does not have the same fragment identifier.

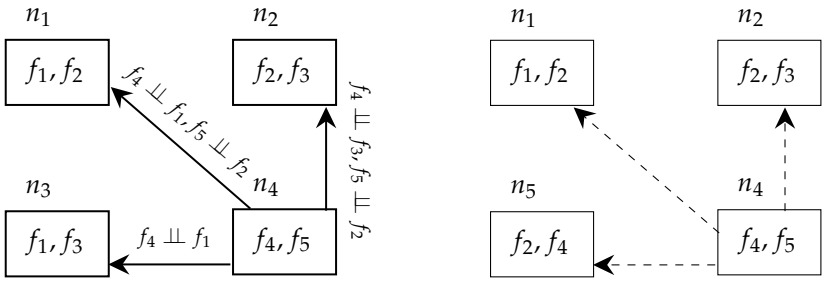
$$Join(n_1, n_2) = \{f_1 \in n_1.\Gamma \mid \exists f_2 \in n_2.\Gamma : f_1 \perp\!\!\!\perp f_2 \wedge f_1.u \neq f_2.u\} \tag{B.2}$$

Example B.2 (Neighbor Ranking)
Consider the fragments in Table B.1b and their assignment to the 4 nodes in Figure B.1a. Note that f_1, f_2, and f_3 are pairwise joinable. We observe that $f_4 \perp\!\!\!\perp f_1$, $f_4 \perp\!\!\!\perp f_3$, and $f_5 \perp\!\!\!\perp f_2$. Assuming we would like to select the least related neighbor of n_4 to shuffle, we apply Equation B.1 and obtain:

- n_1: Since $f_4 \perp\!\!\!\perp f_1$ and $f_5 \perp\!\!\!\perp f_2$, then $r_1 = 2/2 = 1$
- n_2: Since $f_4 \perp\!\!\!\perp f_3$ and $f_5 \perp\!\!\!\perp f_2$, then $r_2 = 2/2 = 1$
- n_3: Since $f_4 \perp\!\!\!\perp f_1$, $f_4 \perp\!\!\!\perp f_3$, $f_5 \not\perp\!\!\!\perp f_1$ and $f_5 \not\perp\!\!\!\perp f_3$, then $r_3 = 1/2 = 0.5$

This results in n_3 being the least related neighbor, and as such it is removed after the shuffle and replaced by a new neighbor n_5 (Figure B.1b).

To compute relatedness in a running system, the nodes exchange the sets of objects and subjects in a compressed representation, such as bitvectors, which can be stored locally for future use.



(a) Joinable fragments at n_4's neighbors ($n_4.N$)

(b) Result of shuffling n_4's least related neighbors ($n_4.N$)

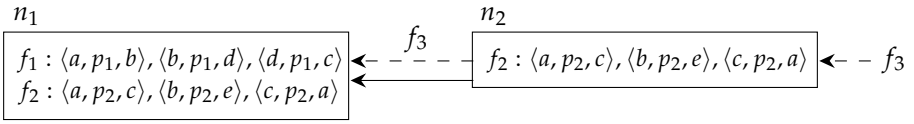
Fig. B.1: Computing the relatedness of n_4's neighbors and shuffling. *Solid arrows denote a connection to a neighbor in list $n_4.N$, and dashed arrows neighbors after a shuffle.*

3.3 Replication of Datasets

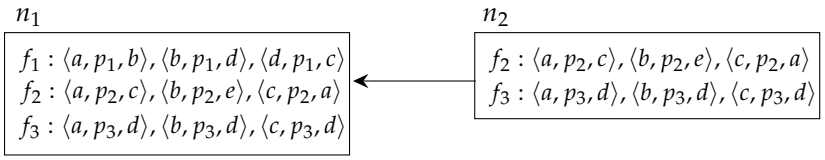
Any node participating in a PIQNIC network can upload a dataset and become its owner node. When uploading a knowledge graph $\mathcal{G}$, a fragmentation function ($\mathcal{F}_P(\mathcal{G})$) is applied to obtain a set of fragments. This set of fragments is then used to create a dataset D.

Allocation of fragments in a PIQNIC network follows a chaining approach, i.e., the owner node passes the fragment on to one of its neighbors, which inserts the fragment into its own local data store and forwards the fragment to one of its neighbors. This continues for a certain number of steps, referred to as *replication factor* (r_f). If a node cannot insert a fragment (for instance because of too little available storage space), it passes the request to one of its neighbors. Lastly, the set of nodes at which the fragment has been inserted is returned to the owner node.

Example B.3 (Allocation and replication of a fragment)
Let us consider nodes n_1 and n_2 in Figure B.2a and fragments f_1, f_2, and f_3 from Table B.1b. Suppose n_2 wants to allocate f_3 with $r_f = 2$. n_2 inserts f_3 into its local datastore, and selects neighbor n_1. f_3 is then forwarded to n_1 with $r_f = r_f - 1 = 1$. f_3 is inserted into n_1's local data store, resulting in Figure B.2b. Since $r_f = 1$, $\{n_1, n_2\}$ is returned to n_2 as the set of nodes in which f_3 has been inserted.



(a) Allocating f_3



(b) States of n_1 and n_2 after allocation

Fig. B.2: Allocating fragment f_3 at node n_2 with $r_f = 2$. *Dashed lines denote the allocation of a fragment, solid lines denote a neighbor relation.*

If a node containing a fragment from $D.F$ fails, the owner will allocate the fragment to another node, ensuring the continued availability of the fragment. If the owner itself fails, another node can take over the task of maintaining availability.

Besides making sure fragments are always available, PIQNIC exposes the following operations, which the owner of a dataset D can execute: (i) add triples to fragments in D, (ii) remove triples from fragments in D, (iii) allocate fragments to further nodes, and (iv) revoke an allocation of a fragment from a node. This update is executed locally on the owner node, after which it forwards the updated fragment to the nodes it is allocated to.

Joining a PIQNIC network can be achieved by knowing an arbitrary node and making it a neighbor. Consider, for instance, a node n_1 wants to join the network via node n_2; n_1 therefore sends a message to n_2, which replies with its neighbors. n_1 will then take over some replicas of these neighbors and gradually become a full member of the network.

4 Query Processing

Any node in a PIQNIC network can issue queries. Query processing follows the basic principle of flooding that is employed in P2P systems [1], i.e., a query is forwarded to a peer's neighbors, which in turn forward it to their neighbors, until a certain Time-To-Live (TTL) value/distance is reached. A query is executed at the node that issued the query, while individual triple patterns may be processed by the node's neighbors, i.e., SPARQL operators, such as `UNION`, are evaluated at the query issuer. In PIQNIC, a SPARQL query q at a node n_i is processed in the following steps:

1. Estimate the cardinality of each triple pattern in q using variable counting. The order in which triple patterns are processed is determined by this estimation, i.e., more selective triple patterns are evaluated first.
2. Evaluate q's triple patterns, starting from n_i's local datastore, over the data accessible via n_i's neighbors by flooding the network using a specified TTL value.
3. Receive partial results from the queried nodes in the network (nodes with no result answer with an empty reply). Partial results are sent directly to the querying node.
4. Compute the final query result by combining the intermediate results of the triple patterns and the remaining operations necessary to complete q.

We use fragment identifiers to avoid querying the same fragment twice on different nodes, i.e., if a fragment with the same identifier has already been queried by a previous node, it will not be queried again. Moreover, if a fragment is available locally, we use that and do not query it again on another node.

Obviously, step 2 can be implemented in different ways. However, before going into details on this aspect, let us first define an identification function (i

in the Definition B.1) to decide whether a fragment is relevant for a particular triple pattern or not. As fragmentation is defined on predicates, we use a predicate-based identification function.

Definition B.7 (Predicate-based identification function)
Let $\mathcal{F}_P(\mathcal{G}_N)$ be a the set of fragments in a network, $f \in \mathcal{F}_P(\mathcal{G}_N)$ be a fragment, tp be a triple pattern and p_{tp} the predicate of tp. A predicate-based identification function $\mathcal{F}_{IP}(f, tp)$ returns true iff $\forall t \in f : p_t = p_{tp}$ or p_{tp} is a variable.

A triple t is said to be a matching triple for a triple pattern tp iff there exists a solution mapping μ s.t. $t = \mu[tp]$, where $\mu[tp]$ is the triple obtained by replacing variables in tp according to μ.

Definition B.8 (Solution mappings [22])
Let U, B, L, V be the set of all URIs, blank nodes, literals, and variables. Then the answer S to a SPARQL query is a set of *solution mappings* s.t. a solution mapping is a partial mapping $\mu : V \mapsto (U \cup B \cup L)$.

We implemented and evaluated three query processing strategies that differ in step 2 of the above description: `Single`, `Bulk`, and `Full`.

Single Strategy

The `Single` approach is inspired by query processing in TPF [22] and Jena ARQ[1]. The triple patterns of a query q are processed sequentially. To process the current triple pattern we use the intermediate results from the node's local fragments (step 1) and previously computed triple patterns. We instantiate the triple pattern with the already known result mapping and send it to the neighbors. This is done for each known solution mapping separately, hence the name of this strategy: `Single`.

Bulk Strategy

Obviously, the `Single` strategy can be improved by sending sets of solution mappings along with a triple pattern throughout the network, instead of individual solution mappings. This is a similar optimization as proposed in [9] to improve TPF query processing. Hence, using the `Bulk` strategy we expect that considerably fewer messages are sent throughout the network. Ideally, all bindings are sent along in a single message. However, for some triple patterns there is a high number of intermediate solution mappings, e.g., query L1 in our evaluation has more than 232,000 solution mappings for variable `?results`. Propagating such a large number of bindings through the

[1] `https://jena.apache.org/documentation/query/index.html`

network might easily become a problem because of the message size. Hence, in such cases, we send the bindings in groups of up to s_m bindings. In our current implementation, we use a default value of $s_m = 1,000$ (empirically determined based on the data and queries used in our experiments).

Full Strategy

In contrast to the other strategies, the `Full` strategy does not include the results of already computed solution mappings in the queries sent throughout the network. Instead, it forwards the triple patterns as defined in the original input query to the neighbors and exploits the fact that this can be done in parallel (instead of sequentially as in the other strategies). However, as this strategy cannot exploit the selectivity of triple patterns if instantiated with solution bindings, in general more data has to be sent through the network. Likewise, more data has to be processed locally at the querying node to compute the final result.

5 Evaluation

We implemented a prototype PIQNIC node[2] in Java 8, using the HDT Java library[3] for the local datastore and extended Apache Jena[4] to support the three query processing approaches discussed in Section 4.

Experimental Setup

We ran our experiments on a server with 4xAMD Opteron 6376, 16 core processors at 2.3GHz, 768KB L1 cache, 16MB L2 cache and 16MB L3 cache each (64 cores in total), and 516GB RAM. To evaluate our approach we used extended LargeRDFBench [10]. LargeRDFBench comes with 13 datasets with altogether over 1 billion triples and was designed to evaluate federated SPARQL query processing engines. LargeRDFBench provides a total of 40 queries divided into four distinct sets: simple (S), complex (C), large data (L), and Complex and large data (CH). However, to enable a more fine-granular analysis, we distinguish the two subsets of S that were originally defined in FedBench [21] but merged together in LargeRDFBench: cross domain (CD) and life sciences (LS).

In our experiments, we varied a broad range of parameters. However, due to space restrictions we do not show all experimental results in this paper

[2] The source code is available at `https://github.com/Chraebe/PIQNIC`

[3] `https://github.com/rdfhdt/hdt-java`

[4] `https://jena.apache.org/`

but focus on a subset. Results of additional experiments are available on our website[5]. For each experiment, we measured the following metrics:

- *Query Execution Time* (QET) is the amount of time it takes to answer a query, i.e., the time elapsed between issuing the query and obtaining the final answer.
- *Completeness* (COM) measures how complete the computed set of answers for a query is; expressed as the percentage of computed answers in comparison to the complete set of answers.
- *Number of Transferred Bytes* (NBT) is the total number of bytes transferred between nodes during query execution.
- *Number of Messages* (NM) is the total number of messages exchanged between nodes during query execution.

Each experiment was run as follows: all queries in the query load were executed 3 times at randomly chosen nodes in the network – the reported measurements represent the averages of the 3 executions.

Experimental Results

Unless stated otherwise, we use the following default values: #Nodes: 200, TTL: 5, Replication: 5%, #Neighbors: 5. We used a timeout of 1200 seconds, i.e., 20 minutes, and the experiments were executed without shuffles during. The average storage space used per node was 1.1GB with 18.1 million triples. The test dataset's size is 61GB with 1 billion triples.

Performance of query execution strategies

`Full` timed out for all queries in groups L and CH, and all but a few queries in C, while `Single` timed out for most queries in the aforementioned groups. Moreover, the queries that timed out for all approaches have a very large number of intermediate results (up to 50 million triples). Such queries are also not supported by state-of-the-art federated query processing approaches [10] and is therefore a general problem that is considered out of the scope of this paper. Therefore, the following discussion focuses on query groups CD and LS – the omitted results can be found on our website[5]. The corresponding query execution times (QET) are shown in Figure B.3.

As we can clearly see, generally `Bulk` performs much better than the other two approaches with respect to execution time. It is not surprising that `Bulk` in general performs better than `Single`, as sending groups of bindings instead of sending each binding separately, considerably reduces communication and computational overhead. While `Full` does perform quite poorly in most cases, it is faster than both `Bulk` and `Single` in rare cases, e.g., LS7. This

[5] `http://relweb.cs.aau.dk/piqnic/`

is due to the fact that some triple patterns have a very low selectivity. In such cases, almost all triples in a fragment are relevant and it is therefore more efficient to download the entire fragment instead of exchanging multiple rounds of messages with large amounts of data.

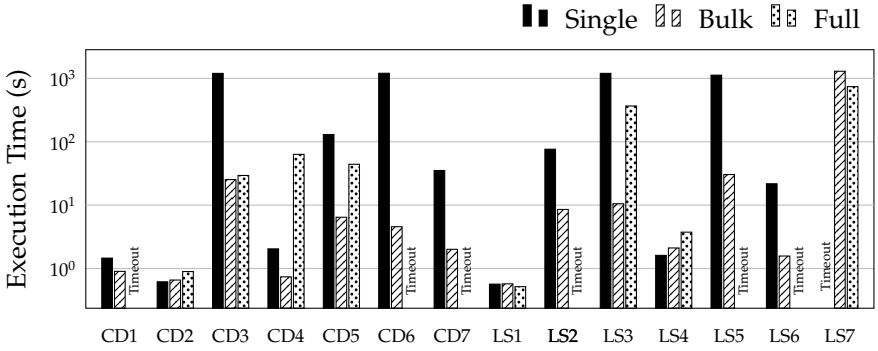


Fig. B.3: QET for `Single`, `Bulk`, and `Full` over queries CD and LS. *Note that the y-axis is in log scale.*

This is evident from Figure B.4, which shows the number of messages sent through the network. Not surprisingly, in all cases `Single` sends more messages throughout the network than the two other approaches. While, expectedly, `Full` is better in this regard than `Bulk`, they are still quite similar in most cases. This is due to the cardinalities of most triple patterns being lower than 1,000, and thus only one message is sent. The queries that did not time out all delivered complete query results (100% completeness).

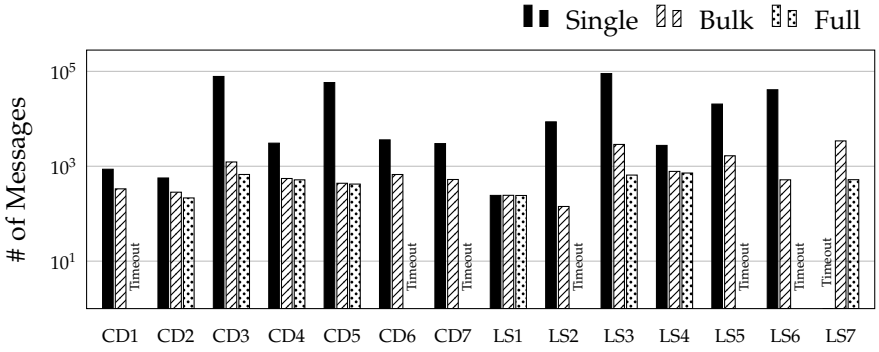


Fig. B.4: NM for `Single`, `Bulk`, and `Full` over queries CD and LS. *Note that the y-axis is in log scale.*

Figure B.5 shows the number of transferred bytes (NTB) for queries in groups CD and LS. Again, we leave out queries that timed out. Not surpris-

ingly, `Full` transfers the most amount of data in all cases. For most queries `Single` and `Bulk` are comparable and the differences negligible.

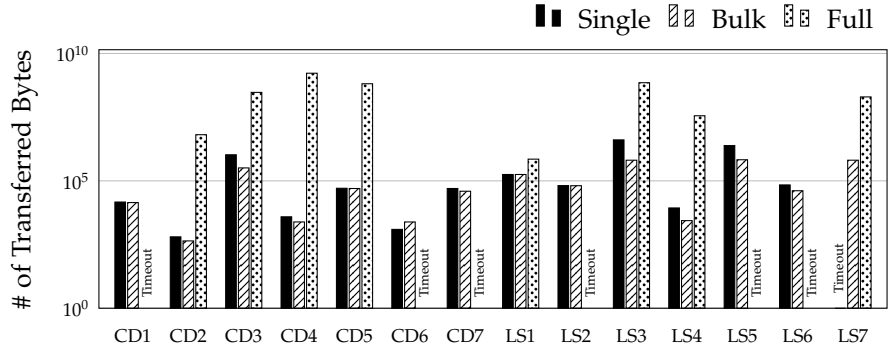


Fig. B.5: NTB for `Single`, `Bulk`, and `Full` over queries CD and LS. *Note that the y-axis is in log scale.*

Robustness of the network

We also evaluated how Piqnic networks perform in the presence of node failures. To test robustness and availability of data, we focused on the `Bulk` strategy and the query sets CD and LS. Figure B.6 shows the average completeness of queries with a varying number of failing nodes. In the experiment, we executed all the queries and noted the completeness, i.e., we gradually killed a randomly selected number of nodes until no nodes were left in the network. The network was given no recovery time after nodes had been killed. The results show the robustness of Piqnic against node failures; the results start with a completeness of 100% and stayed above 90% until less than 60% of the nodes were running. Afterwards, the completeness gradually decreased.

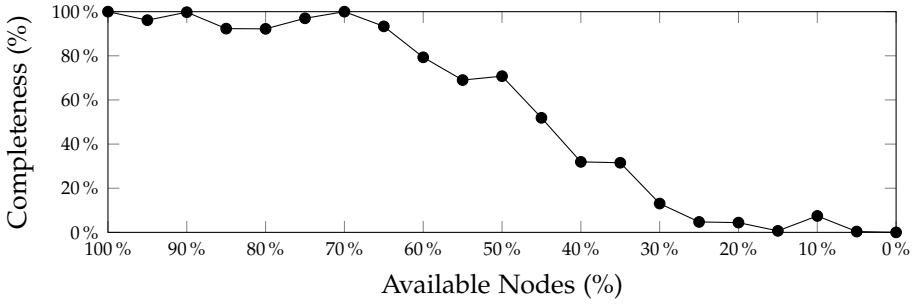


Fig. B.6: COM for queries when varying the number of nodes failures (`Bulk` strategy, no recovery time)

When giving the network recovery time between each run, i.e., allowing all nodes to perform 3 shuffles before the next set of nodes is killed, PIQNIC is able to keep the completeness close to 100% (the lowest was 94,44% due to a single query not being answered) even when 50% of the nodes failed. However, we should mention that query execution time was affected as each node had more fragments to look through. This shows that PIQNIC is able to keep data available through replication at the tradeoff of increased execution time as it is more expensive to find the relevant fragment.

Impact of Time-To-Live

Intuitively, a higher TTL value gives access to a larger part of the network. However, a large TTL value also means sending messages to more nodes. In fact, since we use a flooding technique, the amount of sent messages increases exponentially with the TTL value. To systematically analyze the impact of the TTL value, we compare *COM* and *QET* for three TTL values; 3, 5, and 10. Figure B.7 shows average completeness and execution time for each of the 5 groups of queries in our query load, using the `Bulk` strategy. Even though many of the queries in groups L and CH timed out, they still provided some results before timing out. In general, a TTL value of 3 results in incomplete results for all query groups. We observe that even though a TTL value of 10 gives in total more complete results, the additional query execution time indicates that this might not necessarily be a good tradeoff. Instead, a TTL value of 5 shows almost as complete results with lower execution times.

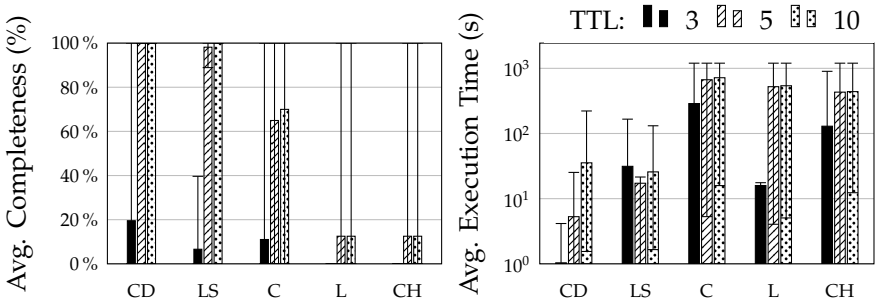


Fig. B.7: Average COM and QET `Bulk` strategy and TTL 3, 5, and 10 (QET log scale)

6 Conclusions

In this paper, we proposed PIQNIC (a P2p clIent for Query processiNg over semantIC data), to process queries over semantic datasets. PIQNIC is inspired by recent advances in decentralized Semantic Web systems as well as P2P

systems in general, and provides an implementation that, in addition to providing query access to vast amounts of data as a client, functions as a server maintaining a local datastore. We presented a general architecture for sharing and processing RDF data in a decentralized manner and customizable approaches for data fragmentation and query processing over a network of nodes. Our experiments show that the `Bulk` strategy provides the best performance on average and that PIQNIC is able to tolerate node failures. As highlighted by one of our experiments, it is not straightforward to find a good balance between completeness, TTL, and query execution time. Moreover, the lack of global knowledge makes it difficult to assess the completeness of a query answer. We will therefore investigate this problem in our future work. Studying the impact of alternative methods for fragmentation and relatedness methods is also part of our future work, as well as processing queries with a high number of intermediate results efficiently.

References

[1] E. Adar and B. A. Huberman, "Free riding on Gnutella," *First Monday*, vol. 5, no. 10, 2000.

[2] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?" in *ISWC*, 2013, pp. 277–293.

[3] M. Cai and M. R. Frank, "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *WWW*, 2004, pp. 650–657.

[4] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF Representation for Publication and Exchange (HDT)," *J. Web Sem.*, vol. 19, p. 22–41, 2013.

[5] P. Folz, H. Skaf-Molli, and P. Molli, "CyCLaDEs: A Decentralized Cache for Triple Pattern Fragments," in *ESWC*, 2016, pp. 455–469.

[6] A. Grall, P. Folz, G. Montoya, H. Skaf-Molli, P. Molli, M. V. Sande, and R. Verborgh, "Ladda: SPARQL queries in the fog of browsers," in *ESWC Satellite Events*, 2017, pp. 126–131.

[7] A. Grall, H. Skaf-Molli, and P. Molli, "SPARQL Query Execution in Networks of Web Browsers," in *DeSemWeb@ISWC*, 2018.

[8] T. Grubenmann, A. Bernstein, D. Moor, and S. Seuken, "Financing the Web of Data with Delayed-Answer Auctions," in *WWW*, 2018, pp. 1033–1042.

[9] O. Hartig and C. Buil-Aranda, "Bindings-restricted triple pattern fragments," in *OTM Conferences*, 2016, pp. 762–779.

[10] A. Hasnain, M. Saleem, A. N. Ngomo, and D. Rebholz-Schuhmann, "Extending largerdfbench for multi-source data at scale for SPARQL endpoint federation," in *SSWS*, 2018.

[11] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi, "Atlas: Storing, updating and querying RDF(S) data on top of DHTs," *J. Web Sem.*, vol. 8, no. 4, pp. 271–277, 2010.

[12] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John, "UniStore: Querying a DHT-based Universal Storage," in *ICDE*, 2007, pp. 1503–1504.

[13] F. Mahdisoltani, J. Biega, and F. M. Suchanek, "YAGO3: A Knowledge Base from Multilingual Wikipedias," in *CIDR*, 2013.

[14] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee, "A Demonstration of the Solid Platform for Social Web Applications," ser. WWW Companion, 2016, pp. 223–226.

[15] E. Marx, M. Saleem, I. Lytra, and A. N. Ngomo, "A decentralized architecture for SPARQL query processing and RDF sharing: A position paper," in *ICSC*, 2018, pp. 274–277.

[16] T. Minier, H. Skaf-Molli, P. Molli, and M. Vidal, "Intelligent clients for replicated triple pattern fragments," in *ESWC*, 2018, pp. 400–414.

[17] P. Molli and H. Skaf-Molli, "Semantic Web in the Fog of Browsers," in *DeSemWeb@ISWC*, 2017.

[18] G. Montoya, C. Aebeloe, and K. Hose, "Towards efficient query processing over heterogeneous RDF interfaces," in *DeSemWeb@ISWC*, 2018.

[19] G. Montoya, H. Skaf-Molli, and K. Hose, "The Odyssey Approach for Optimizing Federated SPARQL Queries," in *ISWC*, 2017, pp. 471–489.

[20] A. Polleres, M. R. Kamdar, J. D. Fernández, T. Tudorache, and M. A. Musen, "A More Decentralized Vision for Linked Data," in *DeSemWeb@ISWC*, 2018.

[21] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran, "Fed-Bench: A Benchmark Suite for Federated Semantic Data Query Processing," in *ISWC*, 2011, pp. 585–600.

[22] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: a low-cost knowledge graph interface for the Web," *J. Web Sem.*, vol. 37–38, pp. 184–206, 2016.

[23] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays," *J. Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

Paper C

Decentralized Indexing over a Network of RDF Peers

Christian Aebeloe, Gabriela Montoya, Katja Hose

The paper has been published in the
Proceedings of the 18th International Semantic Web Conference (ISWC 2019),
pp. 3-20, 2019. DOI: 10.1007/978-3-030-30793-6_1

Abstract

Despite the prospect of a vast Web of interlinked data, the Semantic Web today mostly fails to meet its potential. One of the main problems it faces is rooted in its current architecture, which totally relies on the availability of the servers providing access to the data. These servers are subject to failures, which often results in situations where some data is unavailable. Recent advances have proposed decentralized peer-to-peer based architectures to alleviate this problem. However, for query processing these approaches mostly rely on flooding, a standard technique for peer-to-peer systems, which can easily result in very high network traffic and hence cause high query response times. To still enable efficient query processing in such networks, this paper proposes two indexing schemes, which in a decentralized fashion aim at efficiently finding nodes with relevant data for a given query: Locational Indexes and Prefix-Partitioned Bloom Filters. Our experiments show that such indexing schemes are able to considerably speed up query processing times compared to existing approaches.

Aebeloe C., Montoya G., Hose K. (2019) Decentralized Indexing over a Network of RDF Peers. In: Ghidini C. et al. (eds) The Semantic Web – ISWC 2019. ISWC 2019. Lecture Notes in Computer Science, vol 11778. Springer, Cham.
https://doi.org/10.1007/978-3-030-30793-6_1.
The layout has been revised.

1 Introduction

While there is a huge potential of possible applications of Linked Data and although more and more information is being published in RDF, it is currently not possible to rely on the availability of these datasets. Data providers publish their data as downloadable data dumps, queryable SPARQL endpoints or TPF interfaces, or dereferenceable URIs.

As highlighted in several recent studies [1, 3, 10, 22], it is a huge burden for data providers to keep the data available at all times, making many endpoints often unavailable. Multiple recent studies [1, 4, 8] therefore explored and evidenced the importance of avoiding a single point of failure, e.g. a central server, and maintain a decentralized architecture where data is available even if the original uploader fails through data replication. These approaches, however, either introduce a structured overlay over a peer-to-peer (P2P) network [4], use unstable nodes with limited storage capabilities [8], or make use of inefficient query processing algorithms, such as flooding [1]. Applying a structured overlay to a network of peers restricts peer autonomy as some kind of global knowledge is used to allocate the data at certain peers and to find relevant data for a given query. Apart from general problems, such as finding an optimal way to allocate data at peers, structured overlays need to adjust the overlay when new peers leave or join the network, which may cause problems when a high number of nodes leaves or joins.

Unstructured P2P networks, on the other hand, retain the maximum degree of peer autonomy but with the lack of global knowledge about data placement efficient query processing is considerably more challenging. Hence, unstructured P2P systems typically rely on expensive algorithms, such as flooding, which creates a large overhead and involves exchanging a high number of messages between nodes until the relevant data is actually found and processed. Assuming that each node in the network has N neighbors, flooding results in $\sum_{i=1}^{ttl} N^i$ messages to reach all nodes within a hop distance of ttl (time-to-live value). For example, given a network with $N = 5$ and $ttl = 5$, flooding results in 3,905 messages. Query processing in an unstructured architecture has been addressed previously [8, 9, 16]. However, they either focus on reducing the load on servers by splitting the query processing tasks between multiple clients or rely on unstable nodes with limited storage capabilities. The lack of global knowledge impacts the answer completeness as evidenced in [9], where the average completeness remains under 45%.

In this paper, we do not aim to reduce the server loads or provide users with low-cost but incomplete answers. Instead, to overcome the lack of global knowledge in unstructured architectures and enable efficient query processing, this paper proposes the use of novel indexes, inspired by routing in-

dexes [6], which are tailored for RDF datasets, and provide a node with information about which data its neighbors can provide access to within a distance of several hops. In summary, this paper makes the following contributions:

- Two indexing schemes to determine relevant data based on common subjects and objects: (i) a baseline approach: Locational Index and (ii) an advanced index based on bloom filters: Prefix-Partitioned Bloom Filters.
- Efficient query processing techniques for unstructured decentralized networks using the proposed indexing schemes, and
- An extensive evaluation of the proposed techniques.

This paper is structured as follows: while Section 2 discusses related work, Section 3 describes preliminaries and provides background information. Section 4 proposes the Locational Indexing scheme, followed by Prefix-Partitioned Bloom Filters in Section 5. Query processing is described in Section 6. The results of our experimental study are discussed in Section 7 and the paper concludes with a summary and an outlook to future work in Section 8.

2 Related Work

Although decentralization is not an entirely novel concept, it has gained more and more attention over the last couple of years, especially in the Semantic Web community. The Solid platform [15], for instance, proposes to store personal data in RDF format in a decentralized manner, in so-called Personal Online Datastores (PODs). A POD can be stored on any server at any location, and applications can ask for access to some of its data. This means that data is scattered around the world and that, even if a server fails, most peoples' PODs will still be available. The current focus of Solid, however, is more on protection of private data, whereas we focus on indexing schemes and query processing in decentralized architectures.

To improve availability of data by reducing the load at the servers running SPARQL endpoints, Triple Pattern Fragments (TPF) [22] have been proposed. By processing only triple pattern requests at the servers, query processing load can be reduced and shifted to the clients that then have to process expensive operations, such as joins. Bindings-Restricted TPF (brTPF) [10] further reduces the server load, by bulking bindings from previously evaluated triple patterns, thereby reducing the amount of requests. Other approaches [9, 16] have similarly sought to divide the query processing load among multiple clients, or multiple RDF interfaces [17], in order to speed up query process-

ing. While the previously mentioned approaches greatly reduce the server load, they still have some limitations. Some of these approaches [10, 17, 22] rely on a single server, or a fixed set of servers, that are vulnerable to attacks and represent single points of failure; if the servers fail, all their data will become unavailable, while other approaches [9, 16] rely on unstable nodes with limited storage capabilities. Instead, we focus on architectures in which data is stored, and possibly replicated, in a decentralized and more stable manner, and on reducing the amount of messages sent within such an architecture.

Several decentralized architecures for RDF data are based on structured overlays over a P2P network [4, 13, 14]. These overlays allow to easily identify the nodes that have relevant data to evaluate queries. However, while they have been shown to provide fast query processing, they are vulnerable to churn. This is the case, since each time a node leaves or joins the network, the overlay has to be adapted. This creates a frequent overhead, making such architectures inflexible in unstable environments. Moreover, such structured overlays often impose the placement of data within the network, which is not applicable to the scenario considered in this paper. Therefore, such overlays are not applicable for source selection in our case.

In unstructured networks where the placement of data to the nodes is not imposed, several strategies have been proposed to access the data scattered through the network. Accessing the data may rely on centralized indexes, where one single node is responsible to maintain a full overview of the whole data in the network, and distributed indexes, where nodes are only responsible to provide an overview of the data they store. Centralized indexes represent a single point of failure and it is a challenge to keep the information up-to-date. Diverse approaches, such as [6, 7, 23], represent improvements over the basic flooding algorithm, which distributes the requests to all the nodes in the network, by reducing the number of contacted nodes to answer a query. For instance, routing indexes [6] extend the information that each node includes in its distributed index to include an entry for each of its neighbors and some aggregated information about what data can be accessed by contacting that neighbor within a distance of several hops, locally or by routing the query to a neighbor that has access to such information.

Diverse RDF indexing approaches have been proposed in contexts such as query optimization and source selection [5, 18, 21]. These approaches are mainly based on the structure of the graphs, such as finding representative nodes within the graph, common patterns in the data, or statistical information, such as number of class instances. Our general approach can be combined with many of these approaches, however for our concrete implementation we have focused on summaries based statistical information, since they provide a good tradeoff between index creation time and precision of the indexes. In our case, each node computes its own statistical information, either a locational or PPBF index, and exchanges this information with its

neighbors.

3 Preliminaries

Today's standard data format for semantic data is the Resource Description Framework (RDF)[1]. RDF structures data into triples, which can be visualized as edges in a knowledge graph.

Definition C.1 (RDF Triple)
Given the infinite and disjoint sets U (the set of all URIs/IRIs), B (the set of all blank nodes), and L (the set of all literals), an RDF triple is a triple of the form $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ where s is called the subject, p the predicate, and o the object.

An *RDF graph* g is a finite set of RDF triples. In order to query an RDF graph containing a set of RDF triples, SPARQL[2] is widely used. The building block of a SPARQL query is a *triple pattern*. Triple patterns, like RDF triples, have three elements: subject, predicate, and object, but unlike RDF triples any of these elements could be a variable.

Definition C.2 (Triple Pattern)
Given the infinite and disjoint sets U, B, and L from Definition C.1, and V (the set of all variables), a triple pattern is a straightforward extension of an RDF triple, i.e., a triple of the form $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.

If there is a mapping from the variables in the triple pattern to elements in $U \cup B \cup L$, such that the resulting RDF triple is in an RDF graph, then we say that the triple pattern matches those RDF triples, and that the triple pattern has solutions within the RDF graph. Moreover, in a SPARQL query triple patterns are organized into Basic Graph Patterns (BGPs). A BGP matches only if all the triple patterns within the BGP match. Furthermore, BGP may be combined with other SPARQL operators, such as OPTIONAL or UNION. Even if our approach works well for SPARQL queries with any SPARQL operators, we use examples and descriptions with a single BGP as it makes the explanations simpler and can be naturally extended to SPARQL queries with any number of BGPs.

In an unstructured P2P system, nodes function as both clients and servers. Each node maintains a limited local datastore and a partial view over the network. In the limited local datastore, the node may include one or more RDF graphs. To ease the management of replicated graphs on several nodes, each graph is identified by a URI g. Then, the set of graphs in the local

[1] http://www.w3.org/TR/2004/ REC-rdf-concepts-20040210/
[2] http://www.w3.org/TR/rdf-sparql-query/

repository of node n is denoted as $\mathcal{G}_n$. To keep the network structure stable and up-to-date, peers periodically update their neighbors following certain protocols, such as [23]. The specifics of the partial view over the network may vary from system to system. For example, some systems [1, 7] rank neighbors based on various metrics, e.g., the issued queries or the degree to which the data can be joined.

In this paper, we provide a general approach to identify relevant RDF data within a network. Our approach is based on indexing techniques that are defined independently of specific data placement strategies or network infrastructure. Therefore, our approach can be used in combination with different systems, in particular unstructured P2P networks, which we provide specific details for in Section 6. Furthermore, our general approach to identify relevant RDF data may be used in combination with diverse RDF interfaces to efficiently process queries.

4 Locational Index

Let $P(g)$ be a function that returns the set of predicates within a graph g and $\mathcal{G}_n$ be the set of graphs in the local repository of node n. n's locational index I_L^i(n) then summarizes the graphs that can be reached within a distance of i hops.

Definition C.3 (Locational Index)
Let $\mathcal{N}$ be the set of nodes, $\mathcal{P}$ the set of predicates, and $\mathcal{G}$ the set of graphs, a locational index is a tuple $I_L^i(n)$=$\langle\gamma,\eta\rangle$, with $\gamma : \mathcal{P} \rightarrow 2^{\mathcal{G}}$ and $\eta : \mathcal{G} \rightarrow 2^{\mathcal{N}}$. $\gamma(p)$ returns the set of graphs gs s.t. $\forall g \in gs : p \in P(g)$. $\eta(g)$ returns the set of nodes ns such that $g \in \mathcal{G}_{n_i}$ such that n_i is within i hops from n.

More formally, given that a node n can be described as a triple $n = \langle\mathcal{G}_n, N, u\rangle$, with $\mathcal{G}_n$ being the set of graphs that n stores, N being the set of direct neighbors, and u being a URI that identifies n, a locational index of depth 0 (covering only local graphs) at node n is defined as $I_L^0(n) = \langle\gamma,\eta\rangle$, where:

$$\gamma(p) = \{g \mid g \in \mathcal{G}_n \wedge p \in P(g)\} \tag{C.1}$$
$$\eta(g) = \{n\}, \forall g \in \mathcal{G}_n \tag{C.2}$$

The locational index of depth i for a node n is defined as $I_L^i(n) = \langle\gamma,\eta\rangle$, where:

$$\gamma = I_L^0(n).\gamma \oplus \bigoplus_{n' \in n.N} I_L^{i-1}(n').\gamma \tag{C.3}$$

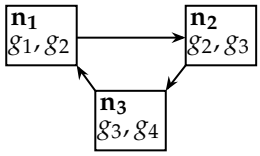
$$\eta = I_L^0(n).\eta \oplus \bigoplus_{n' \in n.N} I_L^{i-1}(n').\eta \tag{C.4}$$

With $(f \oplus g)(x) = f(x) \cup g(x)$ if f and g are defined at x and $f(x), g(x)$ are sets, $(f \oplus g)(x) = f(x)$ if only f is defined at x, and $(f \oplus g)(x) = g(x)$ if only g is defined at x.

Example C.1 (Locational Index)
Consider the graphs in Table C.1a and the nodes and connections in Figure C.1b. Applying Equations C.3-C.4 to create a locational index of depth 2 for node n_1 results in $I_L^2(n_1)$ as shown in Table C.1c.

$\mathbf{g_1}$	$\mathbf{g_2}$	$\mathbf{g_3}$	$\mathbf{g_4}$
⟨a p_1 b⟩	⟨a p_2 d⟩	⟨c p_3 d⟩	⟨b p_2 i⟩
⟨f p_1 a⟩	⟨b p_2 e⟩	⟨g p_3 d⟩	⟨g p_2 b⟩
⟨c p_1 b⟩		⟨e p_3 h⟩	⟨b p_1 j⟩

(a) Datasets and their triples



(b) Connections between nodes

$\mathbf{p}$	$\gamma(p)$	$\mathbf{g}$	$\eta(g)$
p_1	$\{g_1, g_4\}$	g_1	$\{n_1\}$
p_2	$\{g_2, g_4\}$	g_2	$\{n_1, n_2\}$
p_3	$\{g_3\}$	g_3	$\{n_2, n_3\}$
		g_4	$\{n_3\}$

(c) $I_L^2(n_1)$

Fig. C.1: Locational index obtained from a set of datasets and connections

In a real system, locational indexes are built by flooding the network for a specified amount of steps, where each reached node replies with its precomputed locational index.

The nodes that are relevant to evaluate a triple pattern tp, with predicate p_{tp}, are given by $\bigcup_{g \in \gamma(p_{tp})} takeOne(\eta(g))$, if p_{tp} is a URI, or $\bigcup_{g \in range(\gamma)} takeOne(\eta(g))$, if p_{tp} is a variable. *takeOne(s)* returns one element in the set s and it allows for evaluating the triple pattern only once against each graph. Thereby, flooding an entire network can be avoided by only sending triple patterns to relevant nodes. Even in the case of triple patterns with a variable as predicate, the number of requests can be significantly reduced, especially when replicas of graphs are stored at multiple nodes.

Example C.2 (Node Selection with a Locational Index)
Given node n_1 that issues the query, and the triple pattern $tp = \langle ?v_1, p_2, ?v_2 \rangle$, the set of selected nodes from $I_L^2(n_1)$ in Table C.1c is $\{n_1, n_3\}$, since $\gamma(p_2) = \{g_2, g_4\}$, $n_1 \in \eta(g_2)$, and $n_3 \in \eta(g_4)$. The set of selected nodes could be $\{n_2, n_3\}$ because n_2 is also in $\eta(g_2)$, but n_1 may be preferable as it correponds to using the local repository.

5 Prefix-Partitioned Bloom Filters

Building upon the baseline of locational indexes, this section presents Prefix-Partitioned Bloom Filters (PPBFs). The idea is to summarize entities and properties in a graph as a Bloom Filter [2] and rely on efficient bitwise operations on Bloom Filters to estimate if two graphs may have elements in common. We use Bloom Filters since they provide space-efficient bit vectors, and have previously been shown beneficial in reducing information processing for distributed systems [20]. Such knowledge can be used during query processing to reduce intermediate results by only evaluating triple patterns with a join variable over graphs that may have common elements. As such, PPBFs are not complementary to locational indexes, but encode similar information. As we shall see, this further narrows down the list of relevant nodes for a given query.

A Bloom Filter $\mathcal{B}$ for a set S of size n is a tuple $mathcalB = (\hat{b}, H)$, where $\hat{b}$ is a bit vector of size m and H is a set of k hash functions. Each hash function maps the elements from S to a position in $\hat{b}$. To create the Bloom Filter of S, each hash function in H is applied to each element in S, and the resulting positions in $\hat{b}$ are set. o is estimated to be an element of S if all the positions given by applying the hash functions in H to o are set in $\hat{b}$. If at least one corresponding bit is not set, then it is certain that $o \notin S$. However, if all corresponding bits are set, it is still possible that $o \notin S$, meaning a Bloom filter answers the question *is o in S?* with either *no* or *maybe*, rather than *no* or *yes*. The probability of such false positives is given by formula $(1 - e^{-kn/m})^k$ [2]. Furthermore, the cardinality of a set that is represented by a Bloom Filter where t bits are set, can be approximated by the following formula [19]:

$$\hat{S}^{-1}(t) = \frac{ln(1 - t/m)}{k \cdot ln(1 - 1/m)} \tag{C.5}$$

Given two sets s_1 and s_2 and their Bloom Filters $\mathcal{B}_1$ and $\mathcal{B}_2$, with bit vectors of the same size and with the same hash functions, $\mathcal{B}_1 \& \mathcal{B}_2$ approximates the Bloom Filter of $s1 \cap s2$, and $\mathcal{B}_1 | \mathcal{B}_2$ corresponds to the Bloom Filter of $s1 \cup$

$s2$ [12]. Therefore, the number of URIs in two graphs can be approximated using Formula C.5 on the Bloom Filter resulting of applying the bitwise `and` on the graphs' Bloom Filters.

5.1 Partitioning Bloom Filters

In order to have a relatively low false positive probability, the bit vectors should have multiple bits per each possible element. However, in large scale scenarios, e.g., with 500 million distinct URIs, so large bit vectors are not feasible to store for all graphs. If we instead use as few bits as the largest PPBF use, we would still have a high false-positive rate (just above 51% in our experiments). Therefore, instead of having a unique Bloom Filter per graph, we will have a prefix-based partitioning, with a Bloom Filter for each different URI prefix used in the graph.

This has the advantage that not only do most partitions have a low false-positive rate (less than 0.1% in our experiments in Section 7), but even for the partitions that have a high false-positive rate, this is more tolerable since if two URIs have the same prefix, they are more likely to be contained in the same graph, since a prefix typically encodes the domain/source. The prefix of a URI is the URI minus the name of the entity, e.g., the URI `http://dbpedia.org/resource/Auckland` has the prefix `http://dbpedia.org/resource` and the name `Auckland`.

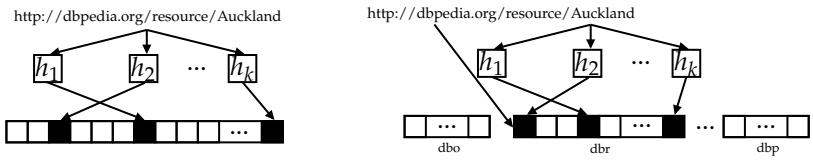
Definition C.4 (Prefix-Partitioned Bloom Filter)
A PPBF $\mathcal{B}^P$ is a 4-tuple $\mathcal{B}^P = \langle P, \hat{B}, \theta, H \rangle$ with the following elements:

- a set of prefixes P,
- a set of bit vectors $\hat{B}$,
- a prefix-mapping function $\theta : P \rightarrow \hat{B}$, and
- a set of hash functions H.

All bit vectors in $\hat{B}$ have the same size. For each $p_i \in P$, $\mathcal{B}_i = (\theta(p_i), H)$, is the Bloom Filter that encodes the URIs' names with prefix p_i. $\mathcal{B}_i$ is called a partition of $\mathcal{B}^P$.

The false positive risk of $\mathcal{B}^P$, is given by its partition with the highest risk. A PPBF for a graph g is denoted $\mathcal{B}^P(g)$ and corresponds to the PPBF for the set of URIs in g. The cardinality of a PPBF is the sum its partitions' cardinalities.

Example C.3 (Prefix-Partitioned Bloom Filter)
Inserting a URI into an Unpartitioned Bloom Filter is visualized in Figure C.2a. Inserting the same URI into a PPBF is visualized in Figure C.2b. Only the name of the entity is hashed, and its hash values set bits only in the partition of its prefix.



(a) Unpartitioned Bloom Filter **(b)** Prefix-Partitioned Bloom Filter

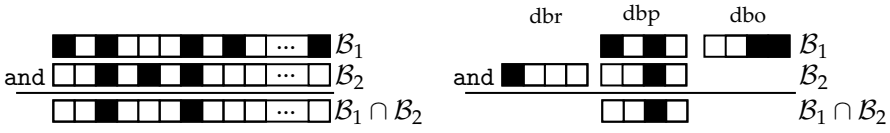
Fig. C.2: Insertion of a URI into an Unpartitioned Bloom Filter and a Prefix-Partitioned Bloom Filter. *dbo, dbr and dbp are short for prefixes from DBpedia,* `ontology`, `resource` *and* `property`, *respectively.*

For simplicity, we say that a URI u with prefix p may be in a PPBF $\mathcal{B}^P$, denoted $u \sqsubseteq \mathcal{B}^P$, iff all the positions given by the hash functions applied to u's name are set in the bit vector $\theta(p)$. Correspondingly, we say that a PPBF $\mathcal{B}^P$ is empty, denoted $\mathcal{B}^P = \emptyset$ iff no bit in any partition in $\mathcal{B}^P$ is set, or it has no partitions. Given that the intersection of two Bloom Filters is given by the bitwise `and` operation, the intersection of two PPBFs is given by:

Definition C.5 (Prefix-Partitioned Bloom Filter Intersection)
The intersection of two PPBFs with the same set of hash functions H and bit vectors of the same size, denoted $\mathcal{B}_1^P \cap \mathcal{B}_2^P$, is $\mathcal{B}_1^P \cap \mathcal{B}_2^P = \langle P_\cap, \hat{B}_\cap, \theta_\cap, H \rangle$, where $P_\cap = \mathcal{B}_1^P.P \cap \mathcal{B}_2^P.P$, $\hat{B}_\cap = \{\mathcal{B}_1^P.\theta(p) \text{ and } \mathcal{B}_2^P.\theta(p) \mid p \in P_\cap\}$, and $\theta_\cap : P_\cap \to \hat{B}_\cap$.

That is, partitions with the same prefix are intersected, while other partitions are not part of $\mathcal{B}_1^P \cap \mathcal{B}_2^P$. The intersection of two PPBFs thereby approximates the common URIs of the graphs that they represent, and Formula C.5 can use used to approximate the number of common URIs.

Example C.4 (Prefix-Partitioned Bloom Filter Intersection)
The intersection of two Unpartitioned Bloom Filters is visualized in Figure C.3a. The intersection of two PPBFs is visualized in Figure C.3b.



(a) Unpartitioned Bloom Filter **(b)** Prefix-Partitioned Bloom Filter

Fig. C.3: Intersection of Unpartitioned Bloom Filters and Prefix-Partitioned Bloom Filters. *dbo, dbr and dbp are short for prefixes from DBpedia,* `ontology`, `resource` *and* `property`, *respectively.*

Building a PPBF for a graph is straightforward. For each URI in the graph, its prefix p identifies the relevant partition $\theta(p)$, and the application of hash functions H to its name determines the bits to set in $\theta(p)$. If θ is not defined for p, it is a bit vector with no bits set, before applying the hash functions.

The intersection of PPBFs can be used at query processing time to prune graphs if they do not have joinable entities for queries with a join variable, even if they contain corresponding URIs. Before execution time, each node can compute PPBFs for the graphs in its local datastore and download PPBFs from nodes in the neighborhood to compute the approximate number of URIs of the graphs in the local datastore in common with the reachable graphs. Any network maintenance strategy could be used to ensure regular updates in order to keep the approximations up-to-date, e.g. periodic shuffles [1].

Definition C.6 (Prefix-Partitioned Bloom Filter Index)
Let $\mathcal{N}$ be the set of nodes, $\mathcal{U}$ the set of URIs, and $\mathcal{G}$ the set of graphs, a PPBF index is a tuple $I_P^i(n) = \langle v, \eta \rangle$ with $v : \mathcal{U} \rightarrow 2^{\mathcal{G}}$ and $\eta : \mathcal{G} \rightarrow 2^{\mathcal{N}}$. $v(u)$ returns the set of graphs gs such that $u \sqsubseteq \mathcal{B}^P(g)$, $\forall g \in gs$. η(g) returns the set of nodes ns such that $g \in \mathcal{G}_{n_i}$ $\forall n_i \in ns$ and n_i is within i hops from n.

5.2 Matching Triple Patterns to Nodes

The relevant nodes for a triple pattern have graphs containing all URIs given in the triple pattern. PPBFs allow for efficiently checking if the graph has these URIs. Similarly to matching triple patterns using the locational index, first, we find the graphs with triples that match the triple patterns in the query. Then, for every pair of triple patterns that share a join variable, we prune graphs that, even if they are relevant for each triple pattern, do not have any common URI. Finally, the set of relevant nodes for each triple pattern is obtained, from these reduced set of relevant graphs, in the same way as when using the locational index.

Algorithm 10 shows how a PPBF index is used to identify the relevant nodes to evaluate the triple patterns in a BGP bgp. Given the PPBF index $I_P^i(n) = \langle v, \eta \rangle$, the graph mapping M_g, which associates triple patterns to set of graphs, is initialized (line 2) for every $tp \in bgp$ as the set of graphs gs such that $u \sqsubseteq \mathcal{B}^P(g)$ for all $g \in gs$ if the set of URIs in tp, $uris(tp)$, is not empty, or $range(I_P^i(n).v)$ otherwise. The function $uris(tp)$ returns the set of URIs in the triple pattern tp. Lines 4-15 select among all the relevant graphs for the triple patterns, computed in line 2, the ones that have some URIs in common for triple patterns with a common join variable. This is, the algorithm selects the graphs that may satisfy the join condition. The PPBFs of the relevant graphs, $\mathcal{B}^P(g_1)$ and $\mathcal{B}^P(g_2)$ are used to approximate if these graphs have any URI in common (line 7), and in such case, these graphs are selected as relevant for tp_1 and tp_2, respectively (lines 8-9). Once all the relevant graphs have been

Algorithm 10 Match BGP To PPBF Index

Input: BGP bgp; Node n; PPBF Index $I_P^i(n) = \langle v, \eta \rangle$
Output: Node Mapping M_n

1: **function** *matchBGPToPPBFIndex*(bgp,n,I_P^i)
2: $M_g \leftarrow$ { (tp, range($I_P^i(n).v$) $\cap \bigcap_{t \in uris(tp)} I_P^i(n).v(t)$) : tp $\in$ bgp }
3: $M'_g \leftarrow$ { (tp, $\emptyset$) : tp $\in$ bgp }
4: **for all** $tp_1, tp_2 \in$ bgp s.t. $vars(tp_1) \cap vars(tp_2) \neq \emptyset$ **do**
5: $G'_1, G'_2 \leftarrow \emptyset$
6: **for all** (g_1, g_2) s.t. $g_1 \in M_g(tp_1)$ and $g_2 \in M_g(tp_2)$ **do**
7: **if** $\mathcal{B}^P(g_1) \cap \mathcal{B}^P(g_2) \neq \emptyset$ **then**
8: $G'_1 \leftarrow G'_1 \cup \{g_1\}$
9: $G'_2 \leftarrow G'_2 \cup \{g_2\}$
10: **if** $G'_1 \neq \emptyset \wedge G'_2 \neq \emptyset$ **then**
11: $M'_g(tp_1) \leftarrow M'_g(tp_1) \cup \{G'_1\}$
12: $M'_g(tp_2) \leftarrow M'_g(tp_2) \cup \{G'_2\}$
13: **else**
14: $M'_g \leftarrow \{(tp, \emptyset) : tp \in bgp\}$
15: **break**
16: **return** { (tp, $\bigcup_{g \in M'_g(tp)}$ takeOne($I_P^i(n).\eta(g)$)) : tp $\in$ bgp }

considered, if any of them have been selected, then the graph mapping M'_g is extended with values for the triple patterns tp_1 and tp_2 (lines 11-12). In other case, it is not possible to find answers for the given bgp and therefore the graph mapping M'_g is initialized again and the loop ends (lines 14-15). Finally, the node mapping M_n is computed in line 16 by using the selected graphs in M'_g and the function $I_P^i(n).\eta(g)$. The function $takeOne(ns)$ returns one of the nodes in ns, if the number of hops between n and the nodes in ns is known, then $takeOne(ns)$ could be implemented to take the node closest to n. In that case, if triple pattern tp is mapped to $\{g_1\}$, $I_P^i(n).\eta(g_1) = \{n_1, n_2\}$, and n_1 is closer to n than n_2, takeOne($\{n_1, n_2\}$)=n_1, and therefore $M_n(tp) = \{n_1\}$ The returned node mapping M_n specifies which nodes should be queried for each triple pattern.

Example C.5 (Node Mapping)
Consider the query Q in Listing C.1 and the set of graphs in Figure C.1a and I_P^i in Table C.1a. Applying Algorithm 10 to Q's bgp results in the set of mappings in Figure C.1b. Besides checking whether each URI in a triple pattern is contained within a PPBF, the algorithm prunes g_4 from the second triple pattern. This is the case, since $p_2, b \sqsubseteq \mathcal{B}^P(g_4)$, but $\mathcal{B}^P(g_4) \cap \mathcal{B}^P(g_3) = \emptyset$. Since g_3 is matched to the third triple pattern, and they join on `?v2`, g_4 is pruned.

```
SELECT * WHERE {
  ?v1 p1 b  .
  b   p2 ?v2 .
  ?v2 p3 ?v3
}
```

Listing C.1: Example query Q.

Table C.1: PPBF index for a set of graphs and the resulting node mappings

(a) $I^i_p(n_1)$

u	$v(u)$	**g**	$\eta(g)$
p_1	$\{g_1, g_4\}$	g_1	$\{n_1\}$
p_2	$\{g_2, g_4\}$	g_2	$\{n_1, n_2\}$
p_3	$\{g_3\}$	g_3	$\{n_2, n_3\}$
b	$\{g_1, g_2, g_4\}$	g_4	$\{n_3\}$

(b) M_n

tp	$\mathbf{M_n(tp)}$
$(?v_1, p_1, b)$	$\{n_1\}$
$(b, p_2, ?v_2)$	$\{n_1\}$
$(?v_2, p_3, ?v_3)$	$\{n_2\}$

Using intersections of PPBFs allows for reducing the set of graphs to consider for a query. This is evident from our experiments (Section 7), where multiple intersections of PPBFs with common prefixes were indeed empty, and less data as a result was transferred between nodes.

6 Query Processing

For simplicity, and in-line with recent proposals on query processing [10, 22], we assume that queries are evaluated triple pattern by triple pattern, and that expensive operations, such as joins, are executed locally at the issuer after executing relevant triple patterns over graphs on nodes identified by our indexes. The evaluation of a triple pattern relies on evaluating the triple pattern against the graphs in the local repositories of a set of nodes. Therefore, we define operators to evaluate a triple pattern using either a locational or PPBF index.

Definitions C.7 and C.8 formally specify operators for retrieving a set of nodes given a triple pattern, using a locational index and PPBF index, respectively. Definition C.9 then specifies an operator for evaluating a triple pattern given such a set of nodes.

Definition C.7 (Locational Selection σ^L)
Let the function $\mathcal{I}(I^i_L(n), p)$ denote the set of nodes that is obtained by using $I^i_L(n)$ to find the relevant nodes to evaluate a triple pattern with predicate p, and $n_1 \in I^i_L(n)$ denote that $n_1 \in \eta(g)$ for some $g \in I^i_L(n).\gamma(p)$. Locational selection for a triple pattern tp on a locational index I^i_L(n) of depth i , denoted

$\sigma^L_{tp}(I^i_L(n))$, is the set $\{n_1 \mid n_1 \in I^i_L(n)\}$ if p_{tp} is a variable, or $\{n_1 \mid n_1 \in \mathcal{I}(I^i_L(n), p_{tp})\}$ otherwise.

Definition C.8 (PPBF Selection σ^P)
Let M_n be the node mapping obtained after applying Algorithm 10 to the BGP which includes tp in the query Q. Given a query Q, the PPBF selection for a triple pattern $tp \in bgp$ and bgp a BGP of Q, obtained using the PPBF index $I^i_P(n)$ of depth i, denoted $\sigma^P_{tp,Q}(I^i_P(n))$, is the selection of the nodes $M_n(tp)$.

Definition C.9 (Node Projection π^N)
Given a set of nodes $\mathcal{N}$, node projection on a triple pattern, denoted $\pi^N_{tp}(\mathcal{N})$, is the set of triples obtained by evaluating tp on the local datastore of the nodes in $\mathcal{N}$. Given the function $\mathcal{T}(n, tp)$, that evaluates tp on n's local datastore, node projection is formally defined as: $\pi^N_{tp}(\mathcal{N}) = \bigcup_{n \in \mathcal{N}} \mathcal{T}(n, tp)$

Implementation Details

The proposed indexes can be used in a broad range of applications. However, motivated by recent efforts in the area of decentralization [1], we show their benefits in the context of an unstructured P2P system. Nodes in an unstructured P2P network often have a limited amount of space for datastores. As such, it does not make sense for a node to download entire graphs. Therefore, we adopt the basic setup outlined in PIQNIC [1]. That is, graphs are split into smaller subgraphs, called *fragments*, based on the predicate of the triples. Each fragment is replicated among multiple nodes. For our setup, we simply view a fragment as a graph and extend the original graph's name with the predicate in the subgraph. Hence, there is no need to encode predicates in PPBFs, which therefore only contain URIs that are either a subject or object in the fragment.

In PIQNIC, query processing is based on the brTPF [10] style of processing queries; triple patterns are flooded throughout the network individually, bound by previous mappings. Locational indexes and PPBFs are useful for avoiding flooding since the query processor can use them to identify precisely which nodes are to be queried. Specifically, a query Q at a node n is processed as follows:

1. Reorder triple patterns in Q based on selectivity. More selective triple patterns (estimated by variable counting) are evaluated first.
2. Evaluate each triple pattern $tp \in Q$ by the following steps:
   (a) Apply either locational selection (σ^L) or PPBF selection (σ^P) on n's local index in order to select the nodes N_{tp} that contain answers to tp.

(b) For each node $n_i \in N_{tp}$, apply node projection (π^N) by evaluating tp on n_i's local datastore.

3. Compute the answer to the query by combining intermediate results from previous steps using the SPARQL operators specified in the query.

Since we use the brTPF style of query processing, the iterative process in step 2 is completed by sending bulks of bindings from previously evaluated triple patterns to the nodes selected by the indexes.

7 Evaluation

To evidence the gains in performance and potential benefits of using our proposed indexing schemes, we implemented locational indexes and PPBF indexes as a module in Java 8[3]. We modified Apache Jena[4] to use the indexes during query processing and extended Piqnic [1] with support for our module in order to provide a fair comparison with an existing system.

7.1 Experimental Setup

Our experiments were run on a single server with 4xAMD Opteron 6373 processors, each with 16 cores (64 cores in total) running at 2.3GHz, with 768KB L1 cache and 16MB L2 and L3 cache. The server has 516GB RAM. We executed several experiments with variations of some parameters, such as ttl, replication factor, and number of neighbors. However, due to space restrictions, we only show the most relevant results in this section. Additional results can be found on our website[5]. The results presented in this section focus on experiments with the following parameters: 200 nodes, TTL: 5, number of neighbors per node: 5. The timeout was set to 1200 seconds (20 minutes). The replication factor was 5%, meaning that with 200 nodes, fragments were replicated on 10 nodes. While, in theory, these parameters should give nodes access to well over 200 nodes, in reality nodes within the same neighborhood often share some neighbors, giving them access to far less nodes. In our experiments, each node had, on average, access to 129.43 nodes. By increasing the TTL value to a sufficiently large number, our indexes could provide a global view, however as nodes are free to join and leave the network, keeping this global view up-to-date can easily become quite expensive. Each dataset was assigned to a random owner, which replicated the fragments across its neighborhood.

[3] The source code is available on our GitHub at `https://github.com/Chraebe/PPBFs`

[4] `https://jena.apache.org/`

[5] Additional results are available on our website at `https://relweb.cs.aau.dk/ppbfs`

We use the queries and datasets in the extended LargeRDFBench [11]. LargeRDFBench comprises 13 different datasets, some of them interconnected, with over 1 billion triples. It includes 40 SPARQL queries, divided into four sets: Simple (S), Complex (C), Large Data (L), and Complex and Large Data (CH). We measure the following metrics:

- *Execution Time (ET)*: The amount of time in milliseconds spent to process a given query.
- *Completeness (COM)*: The percentage of the query answers obtained by a system. To determine completeness, we computed the results in a centralized system and compared them to the results given by the decentralized setup.
- *Number of Transferred Bytes (NTB)*: The total number of transferred bytes between nodes in the network during query processing.
- *Number of Exchanged Messages (NEM)*: The total number of messages exchanged, in both directions, between nodes during query processing.

Queries were run sequentially on random nodes. At most 37 nodes were active at the same time during our experiments. We report averages over three runs.

Storage and Building Times

As we shall see, in most real cases PPBFs outperform locational indexes in terms of performance and data transfer. However, in our experiments, the index creation time was, on average 6,495 ms for locational indexes and 10,992 ms for PPBF indexes. PPBFs used 427MB per node, while locational indexes used 685KB per node. Furthermore, matching triple patterns to nodes is less complex for locational indexes. This means, that for cases where resources are limited, locational indexes might overall be the best choice, given that they still increase performance overall.

7.2 Experimental Results

During all experiments, we compared Piqnic without modifications, Piqnic with locational indexes, and Piqnic with PPBF indexes. The objective is to verify that locational indexes and PPBF indexes can improve query processing, especially for the typically challenging queries.

Performance Gains Using Locational Indexes and PPBF Indexes

Figure C.4 shows ET for query group S. The extended versions with locational indexes and PPBF indexes perform significantly better than the unmodified version for all the queries. Moreover, the version extended with PPBF indexes

is more efficient than the version extended with locational indexes in all cases except queries S6 and S7. For queries S6 and S7, using the PPBF indexes does not allow for pruning any additional nodes than using the locational indexes, and so the slightly larger overhead of testing the graphs for common URIs leads to slightly larger query processing times. However, since all these times are below 100ms, this is negligible compared to the improvements that PPBF indexes provide for other queries. Moreover, for query S9, a locational index does not help. This is due to a triple pattern where all constituents are variables. Because of this, the locational index returns all nodes within the neighborhood, the same set of nodes that PIQNIC uses. The version extended with the PPBF indexes is able to eliminate some of these nodes and thus improve performance.

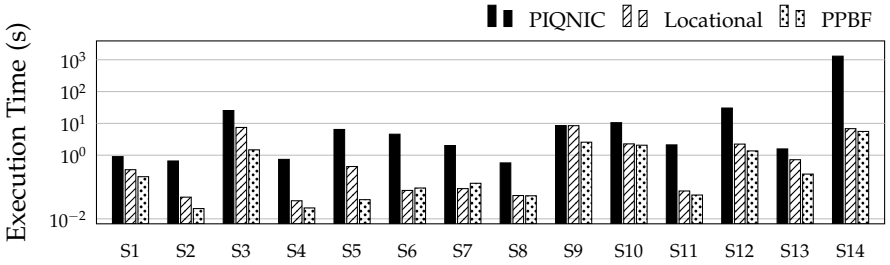


Fig. C.4: ET for PIQNIC, PIQNIC with locational indexes, and PIQNIC with PPBF indexes over query group S. *Note that the y-axis is in log scale.*

Figure C.5 shows ET for query groups L and CH. Generally, queries which were not computable before, are computable with PPBF indexes, alluding to a significantly improved performance. For some queries, such as S14, CH3 and CH6, the improvement was especially significant. Though, some especially large queries could not be processed within the time out. Given enough time, however, we were able to execute these with ET between 2K-10K seconds. The only exception was L5, which has proven to be particularly challenging for state of the art federated processors [11]. Even though we do not show the results for query group C, they showed the same pattern; an improvement in performance using locational indexes, and further improvement using PPBF indexes.

In our experiments, all queries that finished had the same completeness for all approaches. Figure C.6b shows the average COM over query groups. Since the indexes make query processing more efficient, we experienced fewer timeouts, which caused the higher completeness for some query groups.

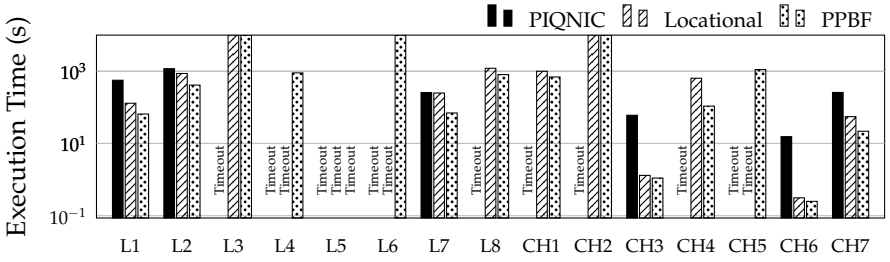
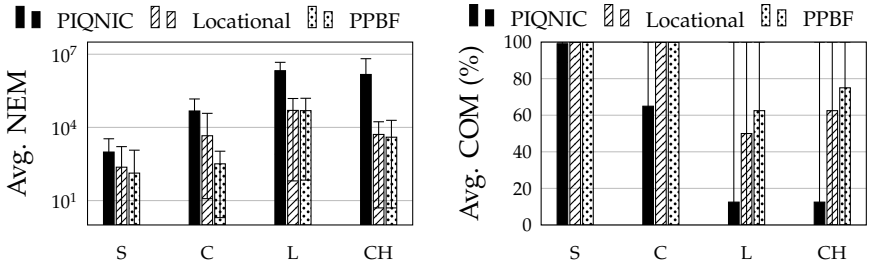


Fig. C.5: ET for PIQNIC, PIQNIC with locational index, and PIQNIC with PPBF indexes over query groups L and CH. *Note that the y-axis is in log scale.*

Index Impact on Network Traffic

One of the major advantages of the indexes presented in this paper is the fact that flooding can be avoided, thus the number of messages exchanged between nodes is significantly reduced.

To evidence the improvement wrt. the network traffic, we measured the amount of messages exchanged between nodes, and the amount of transferred data in bytes, during the execution of the queries in the query load. Figure C.6a shows the number of exchanged messages, averaged over the query groups. As expected, both indexes reduce the amount of messages sent throughout the network by avoiding flooding. This reduction has a stronger impact when the number of messages for the unmodified approach is very high. Furthermore, the PPBF indexes can further reduce the number of nodes queried, thereby further reducing the number of messages sent throughout the network during query processing.



(a) Average NEM

(b) Average COM

Fig. C.6: Average NEM and COM for PIQNIC, locational index, and PPBFs over query groups. *Note that for Figure C.6a, the y-axis is in log scale.*

The amount of transferred bytes during query execution (Figure C.7 for query group S), shows the same general tendency. Using indexes can reduce the number of nodes queried and thereby the amount of transferred bytes

since some fragments are pruned. Furthermore, a PPBF index ensures that only relevant fragments are queried, thus reducing NTB even further. The reduced NTB in practice means, that less time is spent transferring data during query execution. This increases performance, especially for queries with large intermediate results.

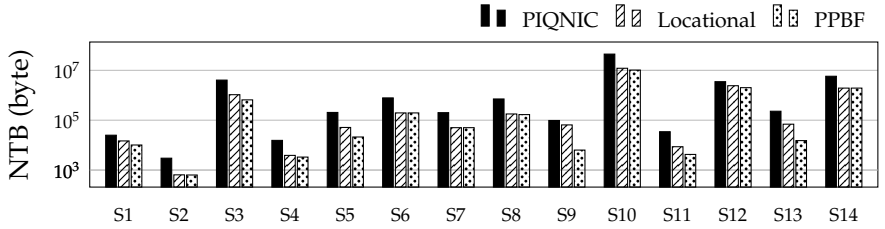


Fig. C.7: NTB for PIQNIC, PIQNIC with locational index, and PIQNIC with PPBF indexes over query group S. *Note that the y-axis is in log scale.*

Impact of Other Parameters

We ran experiments where we varied the time-to-live value, replication factor, and the number of neighbors for each node. For all these experiments, query execution times for the modified approaches were only negligibly affected by the varied network structure, since node matching still only require simple lookups. For the unmodified approach, query execution times were much more affected by the varied network structure. This means, that in terms of completeness, we saw a much greater improvement for the modified approaches than in Figure C.6b, since less queries were completed by the unmodified approach.

8 Conclusions

In this paper, we proposed two schemes for indexing RDF nodes in decentralized architectures: Locational Indexes and Prefix-Partitioned Bloom Filter (PPBF) indexes. Locational indexes establish a baseline, that PPBF indexes extend to provide much more precise indexes. PPBF indexes are based on Bloom Filters and provide summaries of the graph's constituents that are small enough to retrieve the indexes of the reachables nodes without using too much time or space. We implemented both indexing schemes in a module, that could be adapted for use in any decentralized architecture or federated query processing engine. Our experiments show, that both indexing schemes are able to reduce the amount of traffic within the network, and thereby improve query processing times. In the case of PPBF indexes, the

improvement is more significant than for locational indexes. Using PPBFs during join processing to check if a fragment may contain matches given specific values to a join variable could further speed up query processing. This, and studying the impact of using filters with varying sizes, is part of our future work.

References

[1] C. Aebeloe, G. Montoya, and K. Hose, "A Decentralized Architecture for Sharing and Querying Semantic Data," in *ESWC 2019*, 2019, pp. 3–18.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[3] C. Buil-Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche, "SPARQL web-querying infrastructure: Ready for action?" in *ISWC 2013*, 2013, pp. 277–293.

[4] M. Cai and M. R. Frank, "Rdfpeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *WWW*, 2004, pp. 650–657.

[5] Š. Čebirić, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika, "Summarizing semantic graphs: a survey," *VLDBJ*, Dec 2018.

[6] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *ICDCS*, 2002, pp. 23–32.

[7] P. Folz, H. Skaf-Molli, and P. Molli, "Cyclades: A decentralized cache for triple pattern fragments," in *ESWC 2016*, 2016, pp. 455–469.

[8] A. Grall, P. Folz, G. Montoya, H. Skaf-Molli, P. Molli, M. V. Sande, and R. Verborgh, "Ladda: SPARQL queries in the fog of browsers," in *ESWC 2017 Satellite Events*, 2017, pp. 126–131.

[9] A. Grall, H. Skaf-Molli, and P. Molli, "SPARQL query execution in networks of web browsers," in *DeSemWeb@ISWC 2018*, 2018.

[10] O. Hartig and C. B. Aranda, "Bindings-restricted triple pattern fragments," in *OTM 2016 Conferences*, 2016, pp. 762–779.

[11] A. Hasnain, M. Saleem, A. N. Ngomo, and D. Rebholz-Schuhmann, "Extending largerdfbench for multi-source data at scale for SPARQL endpoint federation," in *SSWS@ISWC*, 2018, pp. 203–218.

[12] M. C. Jeffrey and J. G. Steffan, "Understanding bloom filter intersection for lazy address-set disambiguation," in *SPAA 2011*, 2011, pp. 345–354.

[13] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi, "Atlas: Storing, updating and querying RDF(S) data on top of DHTs," *J. Web Sem.*, vol. 8, no. 4, pp. 271–277, 2010.

[14] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John, "UniStore: Querying a DHT-based Universal Storage," in *ICDE 2007*, 2007, pp. 1503–1504.

[15] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee, "A demonstration of the solid platform for social web applications," in *WWW Companion*, 2016, pp. 223–226.

[16] P. Molli and H. Skaf-Molli, "Semantic Web in the Fog of Browsers," in *DeSemWeb@ISWC 2017*, 2017.

[17] G. Montoya, C. Aebeloe, and K. Hose, "Towards efficient query processing over heterogeneous RDF interfaces," in *ISWC 2018 Satellite Events*, 2018, pp. 39–53.

[18] G. Montoya, H. Skaf-Molli, and K. Hose, "The odyssey approach for optimizing federated SPARQL queries," in *ISWC 2017*, 2017, pp. 471–489.

[19] O. Papapetrou, W. Siberski, and W. Nejdl, "Cardinality estimation and dynamic length adaptation for bloom filters," *Distributed and Parallel Databases*, vol. 28, no. 2-3, pp. 119–156, 2010.

[20] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[21] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over linked data," *WWW*, vol. 14, no. 5-6, pp. 495–544, 2011.

[22] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple pattern fragments: A low-cost knowledge graph interface for the web," *J. Web Semant.*, vol. 37-38, pp. 184–206, 2016.

[23] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays," *J. Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

Paper D

ColChain: Collaborative Linked Data Networks

Christian Aebeloe, Gabriela Montoya, Katja Hose

The paper has been published in the
Proceedings of the 30th The Web Conference (WWW 2021), pp. 1385-1396, 2021.
DOI: 10.1145/3442381.3450037

Abstract

One of the major obstacles that currently prevents the Semantic Web from exploiting its full potential is that the data it provides access to is sometimes not available or outdated. The reason is rooted deep within its architecture that relies on data providers to keep the data available, queryable, and up-to-date at all times – an expectation that many data providers in reality cannot live up to for an extended (or infinite) period of time. Hence, decentralized architectures have recently been proposed that use replication to keep the data available in case the data provider fails. Although this increases availability, it does not help keeping the data up-to-date or allow users to query and access previous versions of a dataset. In this paper, we therefore propose ColChain *(COLlaborative knowledge CHAINs), a novel decentralized architecture based on blockchains that not only lowers the burden for the data providers but at the same time also allows users to propose updates to faulty or outdated data, trace updates back to their origin, and query older versions of the data. Our extensive experiments show that* ColChain *reaches these goals while achieving query processing performance comparable to the state of the art.*

Aebeloe C., Montoya G., Hose K. (2021) ColChain: Collaborative Linked Data Networks. In: *WWW 2021: Proceedings of the 30th The Web Conference*.
https://doi.org/10.1145/3442381.3450037.
The layout has been revised.

1 Introduction

The increasing popularity of the Semantic Web and its Web of Data has over the past few years led to a rapid increase in the amount of data published as Linked Open Data (LOD) – spanning a broad range of topics, such as life sciences [12], geography [33], and general knowledge [36]. Such data is made available through either raw data dumps, SPARQL endpoints, or dereferenceable URIs. Yet, the current architecture of the Web of Data requires that we rely on the individual data providers to maintain access to their datasets. While this is a practical and simple solution, it causes several issues that significantly limit the general applicability of technologies relying on the Web of Data as their backbone. As highlighted in several recent studies [4, 7, 34, 35], providing access to such datasets is a significant burden for the data providers, which often results in downtime of public SPARQL endpoints [7, 34]. Another burden for the data provider is the responsibility to keep the data up to date. Current architectures do not support mechanisms to update faulty or outdated data in a community-driven way. Typically, it is the data provider publishing a new version of the entire dataset while taking the old version offline, which then becomes unavailable. However, for many applications it is helpful to access previous versions of the dataset and also trace back updates to ensure data quality [2].

In recent years, decentralized architectures [4, 10, 21] have been proposed to lift some of the burden from the data providers. For instance, RDFPeers [10] uses a structured overlay over a Peer-to-Peer (P2P) network that relies on Dynamic Hash Tables (DHTs) to determine data placement. However, in situations where nodes frequently leave or join the network, such systems have to go through a costly adjustment of the overlay and redistribute the data. Therefore, approaches such as Piqnic [4] rely on unstructured P2P systems instead, where there is no global control over which peer stores which portion of the data. Nevertheless, in both flavors of P2P architectures, replication is used to increase data availability in case of peer failures. However, none of these systems efficiently supports updates, in particular not community-driven ones.

Blockchains [15, 27, 32, 37] are chains of data blocks that represent global ledgers. They rely on the consensus of participating nodes and facilitate updates by adding new blocks to the chain. A new block is linked to the existing chain using hashes to prevent changes without reaching a consensus.

Using blockchain technology for the Web of Data could thus allow users to collaborate on new updates to the data, keep the published data up-to-date, and improve its quality by correcting mistakes in community-driven efforts. However, blockchains typically replicate the chain on all participating nodes [37] to ensure immutability and persistance [32]. While this increases

availability and security, it also requires that every node has to provide a considerably large amount of resources to store multiple large knowledge graphs.

In summary, availability as well as writability issues make it increasingly difficult to trust and rely on the Web of Data since it is sometimes impossible to (i) access the data, (ii) trace back faulty data and updates to the origin, (iii) ensure consistent query answers over longer periods in time by accessing older versions of a dataset, and (iv) update faulty or outdated data. Hence, in this paper we propose ColChain (COLlaborative knowledge CHAINs), a system that increases data availability while enabling users to provide updates and queries over previous versions. In particular, ColChain divides the P2P network into smaller *communities* to increase availability, store updates in chains, and rely on community-wide consensus for updates. Furthermore, with the information in the blockchain, ColChain can trace back updates and access previous versions of the dataset. As such, similar to Wikidata [36] which lets users publish manual updates, ColChain relies on user-provided updates to keep data up to date. In ColChain, however, a consensus is required before an update is applied to the chain, making malicious updates and faulty data less likely. Furthermore, ColChain improves consistency and reproducibility of scientific studies that used query logs such as LSQ [29] in which queries were executed over older versions of well-known datasets. ColChain is *collaborative*, meaning users collaborate on the state of the data by suggesting updates, whereas updates are *consensual*, meaning users form a consensus on each update individually. In summary, this paper makes the following contributions:

- A formal definition of ColChain, a novel decentralized architecture that increases data availability while allowing nodes to trace back updates to the original source as well as propose updates to existing data.
- An approach to process SPARQL queries over previous versions of the datasets published in a ColChain network.
- An extensive evaluation of the performance and overhead of query processing in a ColChain network using a large-scale benchmark (LargeRDFBench [17]).

This paper is organized as follows. Section 2 discusses related work while Section 3 introduces preliminaries. Section 4 presents ColChain and Section 5 outlines consensual updates. Section 6 describes query processing in ColChain. Section 7 then presents experimental results and Section 8 concludes the paper.

2 Related Work

In this section, we discuss approaches to increase the availability of knowledge graphs and the pitfalls of such approaches. Furthermore, we discuss approaches that can accommodate collaborative or consensual updates to data fragments and their shortcomings.

2.1 Client-Server Architectures

SPARQL endpoints remain among the most popular interfaces for querying RDF datasets. SPARQL endpoints are centralized servers that provide an HTTP interface to process SPARQL queries. However, such endpoints are expensive to maintain and experience downtime, leaving the data inaccessible [7, 34]. SaGe [24] increases the availability of the centralized server by suspending queries after a fixed time quantum to avoid long-running queries exhausting the server resources. However, SaGe still processes entire queries on the server, and can thus still suffer downtime.

Several recent approaches [3, 8, 9, 24, 31, 35] attempt to increase the availability of servers by lowering their query processing load. Triple Pattern Fragments (TPF) [35] shift most of the query processing burden from the server to the client, lowering the server load and increasing availability. This is done by ensuring that the server only has to process individual triple patterns, whereas joins and other SPARQL operators are processed by the client. However, while TPF does decrease the load on the server, it incurs in high network usage and high client load, and the performance of TPF depends strongly on factors such as the type of triple pattern and fragment cardinality [18]. Derivatives of TPF [3, 8, 9, 16, 25] therefore aim to further reduce the server load by, for example, sending bulks of previously obtained bindings to the server [16], or increasing the overall throughput by taking advantage of the characteristics of the queries [25]. SPF [3] processes star-shaped subqueries on the server, whereas Smart-KG [8] ships predicate-family partitions to the client. WiseKG [9] combines the approaches presented by SPF and Smart-KG and determines dynamically which strategy is the most cost-efficient for a given subquery. Nevertheless, such systems still rely on centralized servers that are subject to failure.

Federated systems [1, 26, 30, 31] process queries over multiple SPARQL endpoints and make it feasible to exploit the resources of multiple servers during query processing. Nevertheless, since federated systems rely on a fixed set of SPARQL endpoints that are subject to failure, they do not provide a reliable solution to the availability issue. Col-Graph [19] lets consumers create updates to datasets available over federated sources by using `CONSTRUCT` queries to create fragments which they can then change and

expose via SPARQL endpoints. This, however, still requires significant resources on the part of the consumers.

In any case, systems that rely on a central server or a fixed set of central servers do not completely address the availability issue; while reducing the load on the server reduces the chance of server failure, the data will still be unavailable in the event of failure.

2.2 Decentralized Architectures

Decentralized architectures have over the past few years been gaining attention in the Semantic Web community [4, 8, 22, 24, 35]. Several approaches propose to use Peer-to-Peer (P2P) architectures to query RDF data [4, 10, 20, 21]. Such approaches rely on the replication of data over several nodes to increase the availability of the data. Some of these approaches [10, 20, 21] apply a structured overlay such as Dynamic Hash Tables (DHTs) over the network and enforce data placement. While this can be exploited to make query processing relatively efficient, it also makes the networks vulnerable to churn (when nodes frequently leave and join the network) as the overlay then has to be computed again and the data has to be redistributed each time a node leaves or joins the network. Instead, [4] proposes an unstructured network, where connections between nodes are random and replication of data is managed by the data provider. Processing queries in such a network usually relies on flooding the network. This is generally inefficient; however, the query processing performance of such approaches were increased using decentralized indexes [5, 11].

Decentralized systems allow providers to freely upload data to the network. By relying on replication of the data, such systems increase the availability of the data in the event of node failures. However, there is little to no way for consumers to ensure that they have access to up-to-date data, process queries over previous dataset versions, or trace back updates to the original source. Colledge [23] therefore presented a vision of collaborative networks where heterogeneous data providers are connected with consumers. Inspired by this vision, we propose a collaborative network of peers that accommodates consensual updates to data fragments.

2.3 Blockchains

Blockchains [27] define a global ledger of blocks that all nodes store. They rely on the consensus of participating nodes on the state of the ledger. Using blockchains over decentralized knowledge graphs has, to the best of our knowledge, only briefly been researched [15, 32]. However, such systems require the entire chain to be stored on all nodes [37], pack structured data into blocks of a fixed size, and guarantee immutability of the data itself.

Differently, in relational database systems, multiple previous studies [6, 13] have proposed partial solutions to these limitations. BlockchainDB [13] provides a partitioned database layer on top of an existing blockchain, meaning not all nodes store the entire dataset. CAPER [6] defines the chain as a directed acyclic graph in such a way that a node only has to maintain a local view over the entire chain. These approaches, however, only present partial solutions to the limitations mentioned above; they either limit node autonomy, still require the entire chain to be stored on all nodes, or enforce shared relational schema on each node. Furthermore, they view the data within a system as one big dataset rather than several separate datasets owned by different providers.

In this paper, we propose an approach that builds upon blockchains and unstructured P2P systems that structures the network in communities and puts community-based ledgers on top of partitioned knowledge graphs such that each node only stores small subsets of the data and chain.

3 Preliminaries

In this section, we briefly define knowledge graphs followed by the fundamentals of unstructured P2P networks and indexing in such network that ColChain builds upon.

3.1 Knowledge Graphs

The standard format for encoding knowledge graphs is RDF[1]. RDF structures data as triples describing edges in a *knowledge graph*, i.e., a triple consists of a subject, a predicate, and an object.

Definition D.1 (RDF Triple)
Given the infinite and disjoint sets U, B, and L, describing the set of all URIs/IRIs, blank nodes, and literals, an RDF triple is a triple of the form $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$, where s, p, o are called the subject, predicate, and object.

A *knowledge graph* $\mathcal{G}$ is a set of RDF triples. The de facto query language for querying over knowledge graphs is SPARQL[2]. A SPARQL query consists of a set of *triple patterns*.

Definition D.2 (Triple Pattern)
Given the infinite and disjoint sets U, B, L, and V, describing the set of all URIs/IRIs, blank nodes, literals, and variables, a triple pattern is a triple of the form $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.

[1] https://www.w3.org/TR/rdf11-concepts/
[2] https://www.w3.org/TR/sparql11-overview/

We say that a triple pattern p matches a knowledge graph $\mathcal{G}$ iff there exists a mapping from the variables in p to nodes or edges in $\mathcal{G}$ such that applying the mapping to p yields an RDF triple in $\mathcal{G}$. A Basic Graph Pattern (BGP) P is a set of (conjunctive) triple patterns. A SPARQL query consists of BGPs combined with operators. such as `UNION` or `OPTIONAL`. The answer to a BGP P over a knowledge graph $\mathcal{G}$ is given as a *solution mapping*, defined as follows.

Definition D.3 (Solution mapping [35])
Given a BGP P and a knowledge graph $\mathcal{G}$, U, B, L are the sets of URIs, blank nodes, and literals in $\mathcal{G}$, and V is the set of variables in P. A *solution mapping* μ is a partial mapping $\mu : V \mapsto (U \cup B \cup L)$.

Given a triple pattern tp and a solution mapping μ, the notation $\mu[tp]$ denotes the triple (pattern) obtained by replacing variables in tp according to the bindings in μ. A triple t is said to be a *matching* triple to tp if there exists a solution mapping μ such that $t = \mu[tp]$. Furthermore, $dom(\mu)$ returns the *domain* of μ, i.e., the set of variables that are bound in μ and $vars(tp)$ returns the variables in tp.

3.2 ColChain Peer-to-Peer Layer

This section defines the basics of unstructured P2P networks and decentralized indexing [4, 5] that ColChain builds upon.

An unstructured P2P network consists of a set of (possibly heterogeneous) interconnected nodes. Each node in the network maintains a local datastore (a set of knowledge graphs). Furthermore, due to the lack of global knowledge in such a network, each node maintains a partial view over the network, i.e., a set of remote nodes to interact with. By replicating datasets across several nodes, such a network can ensure the availability of the data even if the data provider fails. However, since knowledge graphs today can contain several billions of triples, and nodes in a P2P network are relatively resource restricted, uploaded knowledge graphs are divided into smaller disjoint *fragments*. Fragments can be obtained from a knowledge graph by applying a *fragmentation function*, defined as follows.

Definition D.4 (Fragmentation function)
A fragmentation function F is a function that maps from a knowledge graph $\mathcal{G}$ to a set of fragments, i.e., $F : \mathcal{G} \mapsto 2^{\mathcal{G}}$ such that $\forall f_1, f_2 \in F(\mathcal{G}) : f_1 \cap f_2 = \emptyset$.

An example of a fragmentation function is the very coarse-granular function $F_C(\mathcal{G}) = \{\mathcal{G}\}$ that does not split up the original dataset. Another example is the slightly more fine-granular predicate-based fragmentation function that creates a fragment for each predicate in the knowledge graph [4]. This means that fragments can be adapted to various characteristics of each

knowledge graph. Each fragment has an identifier denoted u_f. To achieve replications of fragments over several nodes, the uploading node selects a neighbor and propagates the fragment in a chain. Furthermore, updates to fragments are only possible if the owner node issues an update and propagates it throughout the network until all replicas are reached.

To speed up query processing, using routing indexes can help limiting the number of nodes to query by identifying which nodes are relevant to a given subquery. As a result, the network overhead is reduced and query processing performance increased. In order to process queries over fragments not in the local datastore, nodes must include indexes for fragments in their distributed index. A distributed index is defined as follows.

Definition D.5 (Distributed Index)
Let n be a node, $\mathcal{N}$ be the set of nodes within a network, $\mathcal{T}$ be the (infinite) set of possible triple patterns, and $\mathcal{F}$ be the (finite) set of fragments that n has access to. A *distributed index* on n is a tuple $I_n = (\nu, \eta)$ with $\nu : \mathcal{T} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^{\mathcal{N}}$. For a triple pattern t, $\nu(t)$ returns the set of fragments in $\mathcal{F}$ that t matches. For a fragment f, $\eta(f)$ returns the nodes on which f is located.

Definition D.6 (Node Mapping)
For any BGP P and distributed index I, there exists a function $match(P, I)$ that returns a *node mapping* $M : T \mapsto 2^{\mathcal{N}}$ where T is the set of triple patterns in P, such that $M(t)$ returns the indexed nodes that have fragments matching t.

To include indexes for remote fragments, nodes have to share particular parts of their distributed index with other nodes. In order to facilitate this, the nodes compute or download *slices* for each fragment not in their local datastore and combine these to form a distributed index. In simple terms, a slice is the part of a distributed index that describes a particular fragment. Formally, a slice is defined as follows.

Definition D.7 (Index Slice)
Let f be a fragment. A *slice* of f, s_f, is a tuple $s_f = (\nu', \eta')$ where $\nu'(t)$ returns f if there exists a triple in f that matches t, and $\eta'(f)$ returns the set of all nodes that contain f in their local datastore. The function $s(f)$ returns the slice of f.

For instance, in [5], the function ν' within the fragment slice definition is implemented as a partitioned bitvector that summarizes the subject and object values of a particular fragment, while η' is implemented as a dictionary that maps fragments to the nodes that have the fragment. Fragment slices can be combined, using the $\oplus$ operator, into a distributed index, Prefix-Partitioned Bloom Filter Index in [5]. For example, slices $s_{f_1} = (\nu'_1, \eta'_1)$ and

$s_{f_2} = (\nu_2', \eta_2')$ are combined into index $I_n = (\nu_1' \oplus \nu_2', \eta_1' \oplus \eta_2')$.[3] The distributed index is then used to check the overlap of fragments during query time to determine which combinations of fragments produce join results. Distributed indexes are composed of the slices of all the accessible fragments, i.e., both locally and remotely available fragments. For a local fragment, the node computes ν' and η', while for a remote fragment, the node retrieves ν' and η' from nodes that have the fragment locally available. Given a set of slices S, the index of S, $I(S)$ can be computed as follows.

$$I(S) = \left(\bigoplus_{s \in S} s.\nu', \bigoplus_{s \in S} s.\eta' \right) \tag{D.1}$$

Query processing in such a setup uses the following general steps:

1. Use indexes to determine which nodes to process each triple pattern over, i.e., retrieve node mappings (Definition D.6).
2. Determine the join order of the triple patterns using, for example, variable counting or cardinality estimations.
3. Process each triple pattern in the determined order, at each step using previously obtained bindings to limit the intermediate results.

4 ColChain

In this section, we provide a brief overview of a ColChain network and the architecture of a ColChain node, followed by a formal definition of ColChain.

4.1 Design and Overview

In conventional blockchains [27], it is up to the nodes to agree on the current status of the chain [37]. To allow for consensual updates to knowledge graphs and to overcome the limitations posed by blockchains (Section 2.3), we make six design choices:

1. Knowledge graphs are divided into smaller fragments according to a specific fragmentation function[4].
2. The network is divided into *communities* of nodes.
3. Rather than a common ledger (henceforth called a *chain*), there is a distinct chain associated with each fragment.

[3] $\oplus$ is defined in [5] as $(f \oplus g)(x) = f(x) \cup g(x)$ if f and g are defined at x; $(f \oplus g)(x) = f(x)$ if f is defined at x; $(f \oplus g)(x) = g(x)$ if g is defined at x.

[4] By default ColChain uses the predicate-based fragmentation function [4]

4. The chains contain updates to fragments, i.e., inserting or removing triples, as well as provenance information.
5. We distinguish between *participants* (storing fragments and validating updates) and *observers* (storing index slices).
6. Participants are allowed to upload new fragments and propose updates to fragments within the community; if there is a community-wide consensus, the updates will be applied.

By dividing the entire network into smaller communities of nodes (defined in Section 4.2) and using fragmentation functions (Definition D.4), we avoid storing large knowledge graphs on each node. Furthermore, replicating fragments on participants in a community ensures that the data will be available even if the uploading node fails. By also letting the nodes participate in or observe any community (participants and observers are defined in Section 4.2) and join multiple communities at the same time, we ensure that nodes have the autonomy to decide for themselves what data to store. Furthermore, by relying on the collaboration of participating nodes, ColChain supports consensual updates to the published fragments. Last, by attaching chains of updates (including timestamps describing the point in time the update was applied) to the fragments themselves, nodes are able to roll-back fragments to a specific point in time and process queries over previous versions.

Since ColChain builds on top of unstructured P2P networks [4, 5], each node has a limited view over the entire network. However, in contrast to such networks, where the local view consists of a set of random neighbors, the local view of a ColChain node is entirely delimited by the communities the node participates in or observes. That is, a ColChain node is connected to all other nodes that participate in or observe at least one of the communities that it participates in or observes. Furthermore, fragments published within a community are replicated on all participants along with their update chain and index slice; observers index the fragments in a community but do not store the fragment itself. Fragments can only be published in one community. ColChain satisfies ACID; that is, it is ensured that transactions are atomic, contain only valid RDF triples, and are serialized in the order of the update chains, and that fragments are stored in permanent data storage.

Any node can create new communities and upload data to them; bootstrapping a ColChain network relies on some of the nodes to create communities and the data providers to upload datasets to these communities (only participants can upload fragments). To discover new communities, nodes ask other nodes for a list of communities they participate in. In order to gain access to the data within a community, a node has to either participate in (and replicate the fragments) or observe (and index the fragments) the community. After a node has joined a community, it expands its local view over the network to include the other participants and observers in the community; when

leaving a community, it shrinks its local view to exclude the other nodes in that community but not in any of the other communities that it participates in or observes. As such, nodes are able to choose the fragments available in their local view and the fragments they store, but they do not choose the nodes that are part of their local view.

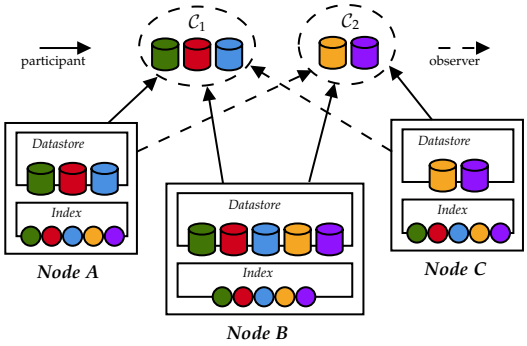


Fig. D.1: A typical ColChain network.

Figure D.1 shows such a typical ColChain network with two communities ($\mathcal{C}_1$ and $\mathcal{C}_2$), each giving access to several fragments. The communities are defined by the nodes that participate in and observe them and the fragments published within them. Figure D.1 visualizes which fragments each community gives access to, as well as participating and observing nodes. In this example, there are three nodes (A, B, and C). Node A participates in $\mathcal{C}_1$ and observes $\mathcal{C}_2$, while node C participates in $\mathcal{C}_2$ and observes $\mathcal{C}_1$. Node B participates in both communities. Since nodes A and C do not participate in both communities, they do not replicate all fragments within the network; thus, to answer queries that require data from both communities, they have to reach out to a node participating in the community they observe. Despite not participating in both communities, nodes A and C can reach complete query answers since they, at least, observe both communities, thus indexing all fragments and participants.

Updates in ColChain are structured in chains of so-called *transactions*; a transaction is a set of bundled operations (either addition or deletion of a triple). While ColChain relies on the consensus of participants to accept transactions, it allows the *owner* nodes of fragments (i.e., the uploading nodes) to enforce transactions without consensus. Owners can also veto transactions proposed by participants. Therefore, we use the signature scheme RSA [28]; when proposing a transaction, the node will attach a signature based on its private key. Participants then validate this signature with the owner's public key to validate ownership over fragments.

Figure D.2 shows the general architecture of a ColChain node (Sec-

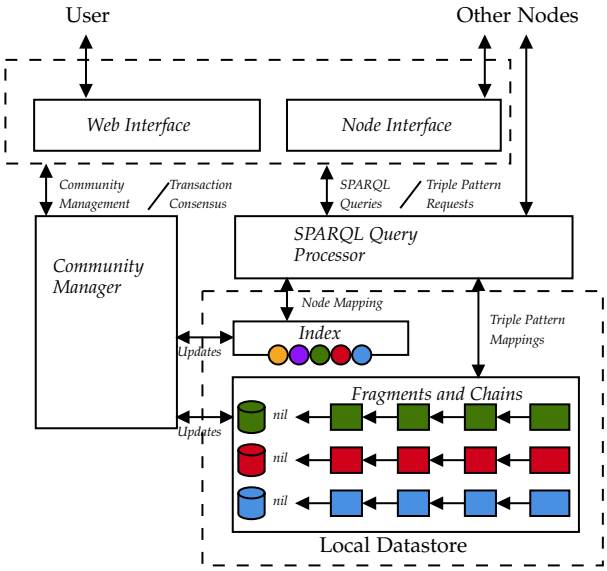


Fig. D.2: Architecture of a ColChain node.

tion 4.2 provides a formal definition). Such a node combines the conventional data storage and blockchain layers in its local datastore. In particular, each fragment in the local datastore is directly associated with a chain of updates called a *secure hash chain*, i.e., each element in the chain is linked to the previous element using its hash value [27]. The SPARQL query processor is able to process full SPARQL queries and triple pattern requests. To process SPARQL queries, ColChain first uses the index to identify the set of relevant nodes and processes the query according to the method described in Section 6. A node has two interfaces; one for users to issue queries, manage communities, and propose updates, i.e., a Web interface, and one for communication between nodes, i.e., the node interface. Each interface communicates with the community manager to manage communities and updates to fragments, and the query processor for SPARQL queries and triple pattern requests.

4.2 Formal Definition of ColChain

A ColChain network consists of a set of nodes $\{n_1, n_2, \ldots, n_n\}$. Each node in such a network contains a local datastore with fragments from zero or more communities.

Definition D.8 (Node)
A node n is a triple $n = (K, a_n, u_n)$ where

- $K = (\kappa_n, \rho_n)$ is a key-pair such that κ_n is n's private key and ρ_n is n's

public key
- a_n is n's address
- u_n is a valid URI and n's unique identifier

Given the definition of a node, we now define the *state* of a node.

Definition D.9 (Node State)
Let n be a node. n's *state* is $\mathcal{S}_n = (\Sigma, S, I_n, \mathcal{M})$ where

- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ and $\forall \sigma_i \in \Sigma : \sigma_i = (\mathcal{X}_i, f_i)$ such that
  - $\mathcal{X}_i$ is a secure hash chain
  - f_i is a fragment
  - All updates represented in $\mathcal{X}_i$ have been applied to f_i
  - All f_j, f_k such that $1 \leq j, k \leq m$ and $j \neq k$ refer to different (unique) fragments in n's local datastore
  - σ_i is said to be an *entry* in n's local datastore
- S is a set of index slices and $\forall \sigma_i \in \Sigma : s(f_i) \in S$
- I_n is n's distributed index, $I_n = I(S)$, consisting of index slices from local and remote fragments
- $\mathcal{M}$ is a set of *metadata* triples

The metadata triples describe a node's local view over the network. These triples include the metadata of all the communities that the node participates in or observes (described in Definition D.11). They include, for instance, other nodes that have joined the community. Since the metadata is structured as a set of triples (i.e., a knowledge graph), it can be stored and managed similarly to fragments. Note, however, that since the number of metadata triples is small, it can be stored in memory. Moreover, versioning and consensus over metadata updates is out of the scope of this paper, and therefore there is no update chain associated to the metadata. Note also that the set of index slices S might contain slices from fragments not included in the node's local datastore. This is the case for fragments in communities observed by the node.

Definition D.10 (Community)
A community $\mathcal{C}$ contains two sets of nodes called participants and observers (defined in Definitions D.12 and D.13) and a set of fragments, each owned by a node, i.e., $\mathcal{C} = (N, F_\mathcal{C}, v, u_\mathcal{C})$ where

- $N = (P_\mathcal{C}, O_\mathcal{C})$ such that $P_\mathcal{C}$ and $O_\mathcal{C}$ are the sets of participants and observers, respectively
- $F_\mathcal{C}$ is a set of fragments
- v is an *ownership-mapping* function such that $\forall f \in F_\mathcal{C} : v(f) = \rho_f$ where $\exists n \in P_\mathcal{C} \bigcup O_\mathcal{C} : \rho_f = \rho_n$
- $u_\mathcal{C}$ is a valid URI and $\mathcal{C}$'s unique identifier

The ownership-mapping function ν maps fragments to the public key of the owner node. ν is used for validating ownership over fragments during consensus (Section 5.2). Given a fragment f, assume that u_f is a valid URI and a unique identifier for f. We now define the state of a community as follows[5].

Definition D.11 (Community State)
Let $\mathcal{C}$ be a community. $\mathcal{C}$'s *state* is $\mathcal{S}_\mathcal{C} = (\Phi, \mathcal{M})$ where

- $\Phi = \{\phi_1, \phi_2, \ldots, \phi_m\}$ and $\forall \phi_i \in \Phi : \phi_i = (\mathcal{X}_i, f_i)$
  - $\mathcal{X}_i$ is a secure hash chain
  - f_i is a fragment
- $\mathcal{M}$ is a set of *metadata* triples such that
  - $\forall n \in P_\mathcal{C} : \langle u_\mathcal{C}, \texttt{cc} : \texttt{participant}, u_n \rangle \in \mathcal{M}$
  - $\forall n \in O_\mathcal{C} : \langle u_\mathcal{C}, \texttt{cc} : \texttt{observer}, u_n \rangle \in \mathcal{M}$
  - $\forall f \in F_\mathcal{C} : \langle u_\mathcal{C}, \texttt{cc} : \texttt{fragment}, u_f \rangle \in \mathcal{M}$ and $\langle u_f, \texttt{cc} : \texttt{author}, \mathcal{C}.\nu(f) \rangle \in \mathcal{M}$

The metadata of a community, $\mathcal{M}$, contains triples detailing all the participating and observing nodes, as well as the fragments within that community and their ownership. Upon joining a community, nodes add these triples to their local metadata,i.e., nodes expand their local view of the network.

Nodes can both participate in or observe a community. Given the definition of a community state, we thus define participants and observers as follows.

Definition D.12 (Participant)
Given a community $\mathcal{C}$ and a node n, n is said to be a *participant* in $\mathcal{C}$ iff for all $\phi = (\mathcal{X}, f) \in \mathcal{S}_\mathcal{C}.\Phi$ it is the case that $(\mathcal{X}, f) \in \mathcal{S}_n.\Sigma$, $s(f) \in \mathcal{S}_n.\mathcal{S}$, and $\forall t \in \mathcal{S}_\mathcal{C}.\mathcal{M} : t \in \mathcal{S}_n.\mathcal{M}$.

A participant is a node that, in its local datastore, contains replicas of the fragments and chains within the community. Note that a node is not limited to participate in just one community. Instead, nodes are free to participate in any community, and as many as they like. By participating in communities, nodes gain more efficient access to the data provided by that community since it is then available from the local datastore. Furthermore, to avoid routing attacks, and to still ensure complete query processing, nodes can also *observe* communities. This gives them access to the data within the community without having to replicate all the fragments.

Definition D.13 (Observer)
Given a community $\mathcal{C}$ and a node n, n is said to be an *observer* in $\mathcal{C}$ iff for all $\phi = (\mathcal{X}, f) \in \mathcal{S}_\mathcal{C}.\Phi$ it is the case that $s(f) \in \mathcal{S}_n.\mathcal{S}$, and $\forall t \in \mathcal{S}_\mathcal{C}.\mathcal{M} : t \in \mathcal{S}_n.\mathcal{M}$.

[5]The prefix `cc :` describes the URI `http://colchain.org/properties#`.

Even though an observer does not store replicas of the fragments itself, it can still access the fragments during query processing by sending requests to community participants. This is done by downloading the index slice of the fragments within the community, and include those in the index. During query processing, the node will then ask participants for the relevant data based on the node mapping provided by the index. Observers are thus able to obtain complete query answers since observing communities requires far less resources than participating.

5 Consensual Updates

In this section, we outline how ColChain enables consensual updates. We provide a formalize updates to fragments and discuss our approach for reaching consensus on proposed updates.

5.1 Updates

In ColChain, there are two types of updates that need to be accounted for; updates to fragments (for participants) and updates to the metadata (e.g., when a node leaves a community). ColChain represents updates to both fragments or metadata as *transactions* that consist of *operations*. To provide access to previous dataset versions, ColChain organizes and stores the transactions done to a fragment over time as a chain. However, since providing access to previous network configurations (e.g., former community members) is out of the scope of this paper, ColChain does not keep the transactions done to the metadata. For ease of presentation, we therefore focus on updates to fragments in this section. Operations can be insertions or deletions and thus updates are in this paper described as deletions followed by insertions. Formally, an operation is defined over a fragment as follows.

Definition D.14 (Operation)
An operation o over a fragment f is a tuple $o = (\alpha, t)$, where α describes whether the operation is an insertion ($\alpha = +$) or deletion ($\alpha = -$) of a triple, and t is the triple that is to be inserted or deleted.

Applying an operation o to a fragment f, denoted $o(f)$, results in a state where the fragment either does not contain the triple (if $\alpha = -$), or contains the triple (if $\alpha = +$). An operation thus represents a change to a fragment in the local datastore. As operations typically occur in sets, we combine updates into *transactions*. Transactions are defined as follows.

Definition D.15 (Transaction)
A transaction γ is a triple $\gamma = (O, f_i, \lambda)$ where $O = \{o_1, o_2, \ldots, o_n\}$ is a set of operations, f_i is a fragment, and $\lambda = (u_n, \alpha_\gamma, \iota)$ is provenance information,

where u_n is the unique identifier of the node proposing the transaction n, α_γ is a signature obtained using n's private key, and ι is a timestamp.

Note that while blocks in conventional blockchains [27] have a fixed size, transactions in ColChain can have varying sizes, i.e., any number of operations. This allows for more efficient storage of updates in situations where updates to knowledge graphs contain different numbers of operations.

Applying all operations in a transaction γ to a fragment f is denoted by $[[f]]_\gamma$. Notice also that a transaction can be applied to the metadata triples. This is denoted $[[\mathcal{M}]]_\gamma$.

Table D.1: Fragment f

Fragment f		
$\langle \mathbf{a}, \mathbf{p_1}, \mathbf{c} \rangle$	$\langle a, p_2, d \rangle$	$\langle b, p_2, d \rangle$
$\langle b, p_3, e \rangle$	$\langle c, p_1, d \rangle$	$\langle c, p_3, a \rangle$
$\langle f, p_4, d \rangle$	$\langle d, p_4, f \rangle$	$\langle e, p_5, g \rangle$

Consider fragment f in Table D.1 and the following transaction:

$$\gamma = (\{(+, \langle a, p_1, b \rangle), (-, \langle a, p_1, c \rangle)\}, f, \lambda)$$

The result of applying the transaction γ to the fragment f, $[[f]]_\gamma$, is the fragment that exchanges the triple $\langle a, p_1, c \rangle$ with $\langle a, p_1, b \rangle$, and can be seen in Table D.2.

Table D.2: The result after applying γ to f, $[[f]]_\gamma$

Fragment $[[f]]_\gamma$		
$\langle \mathbf{a}, \mathbf{p_1}, \mathbf{b} \rangle$	$\langle a, p_2, d \rangle$	$\langle b, p_2, d \rangle$
$\langle b, p_3, e \rangle$	$\langle c, p_1, d \rangle$	$\langle c, p_3, a \rangle$
$\langle f, p_4, d \rangle$	$\langle d, p_4, f \rangle$	$\langle e, p_5, g \rangle$

State transition functions describe how ColChain allows for fragment updates. Transition functions are defined for fragment updates (on participants) as well as fragment slice updates (on both participants and observers). These functions are, upon achieved consensus (described in Section 5.2), triggered on participants and observers. First we define the transition function for the fragments in a state as follows.

Definition D.16 (Fragment Transition Function)
Given a transaction γ, a state $\mathcal{S}_n$ of a node n, and a secure hash function H, the *fragment transition function* is for all $\sigma_i = (\mathcal{X}_i, f_i) \in \mathcal{S}_n.\Sigma$ defined as follows.

$$\tau_\Sigma(\sigma_i, \gamma) = (\tau_\mathcal{X}(\mathcal{X}_i, \gamma), \tau_f(f_i, \gamma)) \tag{D.2}$$

Where

$$\begin{aligned} \tau_{\mathcal{X}}(\mathcal{X}_i, \gamma) = \mathcal{X}_i \cup \{x_{i+1}\} \text{ s.t.} \\ x_{i+1} = (h, \gamma) \\ h = (H(\gamma), y) \\ y = \{H(x_j) \mid j < i\} \end{aligned} \tag{D.3}$$

And

$$\tau_f(f_i, \gamma) = [[f_i]]_\gamma \tag{D.4}$$

The value h in Equation D.3 can be seen as a *header* that contains a hash value of the transaction and links to previous headers (and thus transactions), forming a chain. Since the metadata $\mathcal{M}$ is a set of triples, applying a transaction γ to $\mathcal{M}$ is given by applying $\tau_f(\mathcal{M}, \gamma)$ (τ_f as defined in Equation D.4) on the nodes within the community. Given a transaction γ and a state $\mathcal{S}_n$ of a node n, we therefore define a *slice transition function* as follows.

Definition D.17 (Slice Transition Function)
Given a transaction $\gamma = (O, f_i, \lambda)$ and a state $\mathcal{S}_n$ of a node n such that $s_{f_i} \in \mathcal{S}_n.S$, the *slice transition function* is defined as follows.

$$\tau_S(s_{f_i}, \gamma) = s([[f_i]]_\gamma) \tag{D.5}$$

After a transaction is applied to a fragment, the participants compute a new index slice for the fragment using the slice transition function. Afterwards, participants and observers replace the slice of that fragment with the new slice in their distributed indexes.

5.2 Consensus Protocol

To obtain a consensus on a proposed update, ColChain relies on the participants to actively vote on a transaction. This means that a transaction is not applied until a majority of participants accepts. While this could take some time (since users are active at different times), it prevents faulty or malicious updates from being applied. Furthermore, the data provider can enforce updates to the fragment (i.e., circumvent the majority).

Consensus is reached by a majority of nodes accepting the transaction. The *acceptance* protocol of a participant is defined as follows. Let $verify(\alpha_\gamma, \rho_f)$ be a function that returns `true` iff the signature α_γ matches the public key ρ_f, i.e., if γ was signed by the owner of f, and $validate(\gamma)$ be a function that returns `true` iff the user actively accepts the transaction. The acceptance protocol uses the following steps:

1. Let $\gamma = (O, f, \alpha_\gamma)$, $\mathcal{M}$ be the set of metadata triples of n, $\mathcal{S}_n.\mathcal{M}$, and $\sigma_i = (\mathcal{X}_i, f_i) \in \mathcal{S}_n.\Sigma$ be the unique state entry such that $u_{f_i} = u_f$. Find $\langle u_f, \texttt{cc} : \texttt{author}, \rho_f \rangle \in \mathcal{M}$.
2. If $verify(\alpha_\gamma, \rho_f) = \texttt{true}$, accept γ on σ_i.
3. If $validate(\gamma) = \texttt{true}$, accept γ on σ_i.
4. If $verify(\alpha_\gamma, \rho_f) = \texttt{false}$ and $validate(\gamma) = \texttt{false}$, reject γ on σ_i.

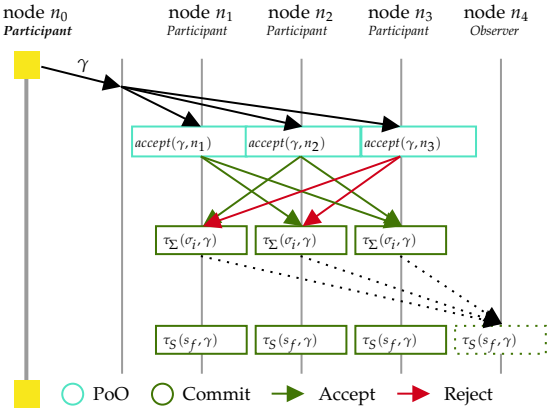


Fig. D.3: Example flowchart of transaction proposition, validation and application.

In a ColChain network, similarly as in [6], transactions are propagated through the community as shown in Figure D.3. First, a node proposes a candidate transaction and calls the acceptance method on each participant in the community.[6] Then, each participant validates the transaction and forwards the result of $accept(\gamma, n)$ to all the other participants. By receiving the acceptance messages from all other participants, nodes are able to, at all times, locally determine when a majority has accepted. In this example, two out of three participants accept the transaction and thus a consensus is reached. After reaching consensus, each participant triggers its state transition functions with the proposed transaction, thereby committing the transaction. Last, the participant that proposed the update forwards the updated index slice to the observers which will, upon receiving an accept message from a majority of nodes, update their index accordingly. This way, changes to the fragments within the community can be collaboratively determined.

Relying on a majority of the users to actively accept transactions creates a practical limitation on the number and size of transactions a user can vote on in reality. This limitation could be addressed by using different consensus protocols in different situations that make active participation of users more scalable, e.g., by detecting non-trivial changes in large updates or malicious

[6] community participants and their addresses are available in the node's metadata

updates. This is part of our future work (Section 8).

6 Query Processing

In line with the state of the art [4, 5], a ColChain node processes SPARQL queries by requesting reachable fragments that match the query's triple patterns, while operations like joins are evaluated locally on the querying node. In line with [4, 16] we attach the bindings from previously evaluated triple patterns to the requests when evaluating subsequent triple patterns. This is to reduce the cardinality of the retrieved fragment, i.e., sizes of intermediate results. Furthermore, by using the node's distributed index (as defined in Definition D.5), it is possible to limit the number of nodes to query for each triple pattern and avoid having to flood the network, i.e., forwarding requests through several layers of neighbors, which is the basic strategy in unstructured P2P networks [4]. Similarly to [4], a ColChain node has a limited view over the entire network. However, differently from [4], this view is not random but determined by the communities the node has joined.

Recall in Definition D.6 the function $match(P, I)$ for a BGP P and index I. The function returns a node mapping M that, given a triple pattern tp, $M(tp)$, returns the set of nodes N with relevant fragments, i.e., fragments that tp matches. Each node n contains a *selector* function that formalizes how triple pattern requests are processed over an entry in n's local datastore. Since ColChain bulks bindings from previously evaluated triple patterns, we define a selector function in line with [16] that takes these bindings into account. Furthermore, to allow for queries over previous dataset versions, we extend the selector function to also take into account a timestamp ι. Let $\sigma_i = (\mathcal{X}_i, f_i)$ be an entry in n's local datastore. $\iota(\sigma_i)$ denotes the fragment obtained by reversing the operations in the transactions encoded within $\mathcal{X}_i$ with a timestamp greater than or equal to ι, i.e., γ_j with $\gamma_j.\lambda.\iota \geq \iota$, i.e., the fragment that was available at ι. The selector function is defined as follows.

Definition D.18 (Selector Function)
Given a node n, a triple pattern tp, a finite set of distinct solution mappings Ω, and a timestamp ι, the fragment-based bindings-restricted triple pattern selector for tp, Ω, and ι, denoted $s_{(tp,\Omega,\iota)}$, is for every entry σ in n's local datastore defined as follows.

$$s_{(tp,\Omega,\iota)}(\sigma) = \begin{cases} \{t \in \iota(\sigma) \mid t \in \iota(\sigma) \wedge \exists \mu : t = \mu[tp] & \text{if } \Omega = \emptyset \\ \{t \in \iota(\sigma) \mid t \in \iota(\sigma) \wedge \exists \mu : t = \mu[tp] \wedge & \\ \qquad \exists \mu' \in \Omega : \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

In line with the state of the art, we apply pagination [16, 35], i.e., we group the results into reasonably sized pages (e.g., 100 results per page in [35]) to

avoid excessive data transfer. For ease of presentation though, we focus on the case where all answers fit into a single page. Given a node n, a triple pattern tp, a set of solution mappings Ω, and a timestamp ι, we define two additional functions that call the selector function (Definition D.18) to process triple pattern requests: $n^c(tp, \Omega, \iota)$ and $n^p(tp, \Omega, \iota)$.

- $n^c(tp, \Omega, \iota)$ returns a cardinality estimation of the result of invoking $s_{(tp,\Omega,\iota)}$ on n.
- $n^p(tp, \Omega, \iota)$ returns the result of invoking $s_{(tp,\Omega,\iota)}$ on n.

Note that in some cases nodes will process triple patterns over fragments available in the local datastore. In this case, the returned node n from the node mapping is the local node, and the node does not actually perform a request to itself; rather it just processes the triple pattern locally and forwards the result to the query processor.

Algorithm 11 Evaluate a BGP on a ColChain node

Input: A BGP $P = \{tp_1, \ldots, tp_n\}$; a node n; a timestamp ι; a set of solution mappings Ω; a node mapping M
Output: A set of solution mappings

1: **function** *evaluateBGP*(P,n,ι,$\Omega = \emptyset$,$M = \emptyset$)
2: **if** $M = \emptyset$ **then** $M \leftarrow match(P, \mathcal{S}_n.I_n)$;
3: **for all** $tp_i \in P$ **do**
4: $cnt_i \leftarrow \Sigma_{n_1 \in M(tp_i)} n_1^c(tp_i, \Omega, \iota)$;
5: **if** $cnt_i = 0$ **then return** $\emptyset$;
6: $tp_\epsilon \leftarrow tp_k$ where $tp_k \in P$ and $cnt_k \leq cnt_j \forall tp_j \in P$;
7: $\phi^\epsilon \leftarrow \bigcup_{n_1 \in M(tp_\epsilon)} n_1^p(tp_\epsilon, \Omega, \iota)$;
8: $\Omega^\epsilon \leftarrow \Omega \bowtie \{\mu \mid dom(\mu) = vars(tp_\epsilon)$ and $\mu[tp_\epsilon] \in \phi^\epsilon\}$;
9: $P' \leftarrow P \setminus \{tp_\epsilon\}$;
10: **if** $P' = \emptyset$ **then return** Ω^ϵ;
11: **return** $evaluateBGP(P', n, \iota, \Omega^\epsilon, M)$;

Given a BGP P and timestamp ι specifying which fragment versions to process P over, Algorithm 11 defines a recursive algorithm to process P over the fragments reachable for a node n (i.e., covered by n's index). First, we obtain the node mapping (Section 3, Definition D.6) from n's index (Section 3, Definition D.5) in line 2. Recall that a node mapping given a triple pattern tp, $M(tp)$, returns a set of nodes that contain relevant fragments to tp. Using the previously defined function $n^p(tp, \Omega, \iota)$, we then obtain a cardinality estimation for each triple pattern by summing the cardinality estimations obtained by each node specified by the node mapping we found in the previous step (lines 3-5). The triple pattern with lowest cardinality estimation is then

evaluated first by sending it to all sources specified by the node mapping to obtain the bindings for this triple pattern (line 7). The output results from previous triple patterns are then joined with the bindings just obtained (line 8). Last, the evaluated triple pattern is removed from the BGP (line 9) and, if there are any remaining triple patterns, the algorithm is called recursively with the new BGP and set of bindings (line 11).

7 Experimental Evaluation

We compare ColChain with state-of-the-art decentralized architectures for sharing and querying semantic data in terms of performance. Moreover, we investigate the impact of community sizes on ColChain's performance and updates and study ColChain's performance when processing queries against previous versions.

7.1 Implementation Details

A prototype of ColChain was implemented in Java 8[7]. Both the Web interface and node interface (Figure D.2) are implemented as Java 8 Servlets using Jetty[8]. The query processor is implemented as an extension to Apache Jena[9] and can process SPARQL queries that Jena can process (e.g., with `UNION` and `OPTIONAL`). Our implementation uses Prefix-Partitioned Bloom Filter indexes from [5] as distributed index (I_n) and the predicate-based fragmentation function from [4]. The fragments themselves are stored as separate HDT files [14] that allow for efficient processing of triple patterns. The chains of transactions are stored separately from the fragments and if possible in main memory.

7.2 Experimental Setup

Datasets and Queries. We use the datasets and queries from the extended LargeRDFBench [17] benchmark, which comprises 13 interconnected datasets that totals over 1 billion triples. LargeRDFBench includes 40 SPARQL queries divided into 5 different categories: Simple (S), Complex (C), Large Data (L), and Complex and Large Data (CH). We generated chains of 1, 10, and 100 transactions for each fragment. Each transaction has a random number of operations between 10 and 100. We assessed the correctness of query answers by comparing the output to the results obtained using the same data and

[7]The prototype is available at `https://github.com/ColChain/ColChain-Java`.
[8]`https://www.eclipse.org/jetty/`
[9]`https://jena.apache.org/`

queries in a standard triple store. In our experiments, all queries that finished returned complete results.

Experimental Setup. We ran our experiments on a network with 128 nodes. We created 4004 fragments from the LargeRDFBench data in total and 200 communities each with between 10 and 31 fragments (randomly assigned). Nodes are assigned randomly to participate and observe each of the communities. The nodes observe all communities they do not participate in, i.e., all the fragments are reachable from all the nodes, and therefore complete query results can be obtained during query processing. We compare the following systems: (i) an unstructured P2P network with flooding [4] (PIQNIC), (ii) the same system with decentralized indexes [5] (PIQNIC$_{PPBF}$), and (iii) COLCHAIN using the same indexes as PIQNIC$_{PPBF}$. In PIQNIC, fragments are replicated randomly throughout the network and connections are random. We match the replication factor in PIQNIC and PIQNIC$_{PPBF}$ (i.e., the number of replicas per fragment) with the number of participants in each community for COLCHAIN. For each experiment, we ran the 40 queries in the benchmark sequentially at three random nodes and report the average over the three runs.

Parameters. To test how the characteristics of a COLCHAIN network affect performance and update overhead time, we ran experiments with 5, 10, and 20 participants per community. Moreover, to assess COLCHAIN in a more realistic setup with varying community sizes, we ran experiments using a Zipfian distribution of the community sizes, i.e., the most popular community has all 128 nodes as participants, after which the nth most popular community has $128/n$, or at least one, participants. In the versioning experiments, we assessed how scalable COLCHAIN is for different chain lengths (i.e., the number of transactions to roll back for a single query) when processing queries over previous versions. Specifically, we processed queries with chain lengths of 1, 10, and 100.

Hardware Configuration. We ran, for each P2P system, 128 nodes on a virtual machine (VM) with 128 vCPU cores with a clock speed of 2.5GHz, 64KB L1 cache, 512KB L2 cache, 8192KB L3 cache and 2TB main memory. Since all clients are run concurrently on the same machine, they are limited to use 1 core and 15GB memory, and the connection between them is limited to 20 MBit/sec.

Evaluation Metrics. We measure the following metrics:

- *Query Execution Time (QET)*: The amount of time (in milliseconds) it takes to obtain the full query answer.
- *Query Response Time (QRT)*: The amount of time (in milliseconds) it takes to obtain the first query answer.
- *Update Overhead Time (UOT)*: The amount of time (in milliseconds)

elapsed from when an update is proposed to when it is committed on all targets (assuming a consensus is reached instantly).
- *Version Materialization Time (VMT)*: The amount of time (in milliseconds) it takes to materialize previous versions of requested fragments.
- *Number of Exchanged Messages (NEM)*: The number of messages exchanged between nodes.
- *Number of Transferred Bytes (NTB)*: The amount of data transferred (in bytes) between nodes.

Software Configuration. The number of results that each system is allowed to attach to requests is set to $|\Omega| = 30$ (Section 6), and the page size to 100 (i.e., each system can only send 100 results at a time). The timeout is set to 1,200 seconds (20 minutes). For PIQNIC, we use the following values (as recommended by [4]). Time-to-Live (number of hops): 5, Number of Neighbors: 5.

7.3 Experimental Results

Due to space limitations, we will only show the most interesting experimental results in this section. The full experimental results are available at `https://relweb.cs.aau.dk/colchain`.

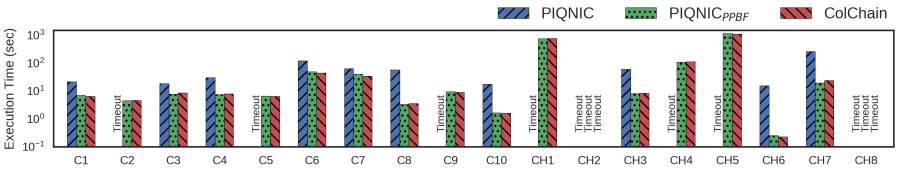
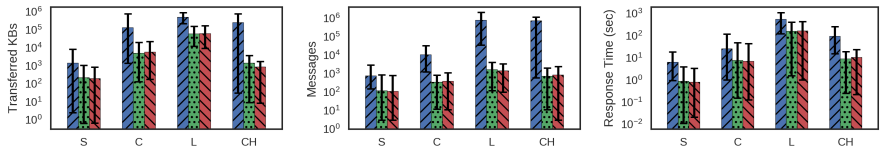


Fig. D.4: Execution time for queries in the C and CH categories for PIQNIC, PIQNIC$_{PPBF}$, and COLCHAIN *(y-axis in log scale)*.



(a) Transferred KBs **(b)** Messages **(c)** Response Time

Fig. D.5: Number of transferred bytes (NTB) in KBs, number of exchanged messages (NEM), and response time (QRT) for each approach over the different query categories excluding queries that timed out *(y-axis in log scale)*.

Query Processing Performance. Figure D.4 shows the query execution time (QET) for each system over the queries in the C and CH query categories. COLCHAIN has similar performance to PIQNIC$_{PPBF}$. This is due to the fact that

the characteristics of a ColChain network are similar to that of a Piqnic network; in our setup, the number of participants in a community was matched to the replication factor in Piqnic. As such, since ColChain and Piqnic$_{PPBF}$ use the same indexing scheme, they have similar overall performance. Only a few queries, such as C3 and CH7, show a slight difference in execution time for ColChain and Piqnic$_{PPBF}$. This is due to the particular topology of the networks and that the randomly chosen nodes have different fragments available in the local datastore, e.g., for CH7, one of the Piqnic$_{PPBF}$ nodes was able execute the query issuing just 2,058 triple pattern requests, while all the ColChain nodes required at least 2,423 triple pattern requests to gather the fragments that were not available locally at the issuing node. The decerease in execution time this caused the particular node to have was significant enough to still be visible even with the value averaged over three separate nodes. Furthermore, ColChain and Piqnic$_{PPBF}$ have better query processing performance across all queries compared to Piqnic. This is consistent across not just the C and CH query categories, but all remaining query categories in our experiments as well. ColChain and Piqnic$_{PPBF}$ timed out for 5 out of the 40 queries, and Piqnic timed out for 14 out of the 40 queries. The timeouts were in the L and CH query categories and were queries for which even federated systems timed out [17].

Figure D.5 shows, for each query category and over each compared system, the number of transferred bytes (Figure D.5a), number of exchanges messages (Figure D.5b), and the query response time (Figure D.5c). Both ColChain and PIQNIC$_{PPBF}$ generate significantly less network load than PIQNIC. This is due to the ability to accurately determine which other nodes to query for specific data, thus removing the need to flood the network. A similar trend can be observed in Figure D.5b where ColChain and PIQNIC$_{PPBF}$ incur significantly fewer exchanged messages than Piqnic. As a result, ColChain is able to maintain a comparatively low response time similar to PIQNIC$_{PPBF}$'s (Figure D.5c).

Overall, our results clearly show that ColChain's performance is comparable to state-of-the-art systems without incurring in any additional network load. This was expected since improving query processing performance was not part of our contributions, and emphasizes that the additional functionality of consensual updates and query processing over previous versions does not decrease the performance when comparing to state-of-the-art systems.

Impact of the Size of the Communities. Figure D.6 shows the query execution time (QET) for each query in the S query category over each community size. It is clear that in most cases the networks with larger community sizes show slightly better performance overall. This is due to the fact that communities with larger numbers of participants have more fragments replicated at each node, and therefore executing a query naturally incurs in a lower num-

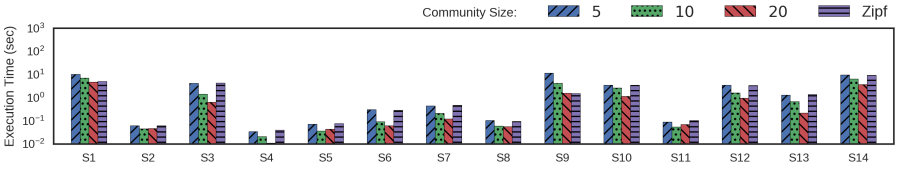
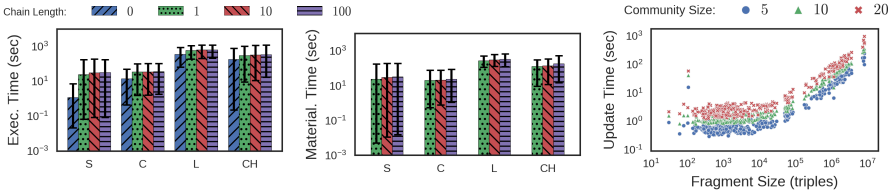


Fig. D.6: Execution time for queries in the S query category for different community sizes *(y-axis in log scale)*.



(a) Execution Time

(b) Materialization Time

(c) Update Overhead Time

Fig. D.7: Execution and materialization time for the versioning experiments excluding queries that timed out *(y-axis in log scale)* and update overhead time over different community sizes *(log scale)*.

ber of requests. In our experiments, the network with five participants per community has between 47 and 332 fragments per node, while the network with 20 participants per community has between 408 and 885. There were a few exceptions, e.g., queries S2 and S5, but this comes naturally from the fact that the additional replicas were placed randomly and were not necessarily located at the node issuing the query. In all cases, the network with five participants in each community is the slowest, and for some queries this margin is quite significant (e.g., up to 7 times slower for S9 compared to the setup with 20 participants in each community). These results are consistent across all query categories and show that there is a correlation between the sizes of the communities and the performance. Finally, we have also considered a configuration where the number of the participants per community varies according to a Zipfian distribution. For this configuration, the results show that query processing is less efficient than in the setup where all communities have 10 or 20 participants. This is the case since in a Zipfian distribution there will on average be fewer participants per communities, resulting in fewer fragments locally available on each node. In fact, in our experiments, nodes have between 19 and 258 fragments available, which is significantly less than in any other setup. Still, the performance is similar to the setup with 5 participants. Moreover, for a few queries such as S1 and S9, the performance is close to the setup with 20 participants. This is a result of many of those fragments being part of a popular community, i.e., there is a high like-

lihood of such fragments being available in the local datastore of the node issuing the query.

Cost of Processing Queries over Previous Dataset Versions. In order to study ColChain's performance when processing queries against previous dataset versions, we created update chains with 1, 10, and 100 transactions per fragment. We applied these updates to the fragments and saved only the latest version in the local datastore. All queries were evaluated over the initial version of the datasets by rolling back the entire chain and computing the initial version during query processing. Figures D.7a-D.7b shows, for each query category the query execution time (Figure D.7a) and the version materialization time (Figure D.7b). Our experiments show that while processing queries over a previous dataset version clearly takes more time, the length of the chain (i.e., number of transactions to roll back) actually has very little impact. This is a sub-linear effect since it is not necessary to materialize each version in between the current version and the target version; rather it is sufficient to simply combine all the operations in the transactions and materialize the target version by rolling back those operations on the current version. Moreover, it is clearly visible that the materialization time (Figure D.7b) does not significantly vary with the number of transactions; instead, it is mostly impacted by the query category. One of the main differences between the different query categories is the number of transferred bytes. In particular, while the S query category requires at most the transfer of 856 kilobytes, the L query category requires at most the transfer of 154 megabytes. Clearly, the materialization time is the main component of the query execution time (cf. Figure D.7a and Figure D.7b), where up to 94% of the materialization time is spent in decompressing the fragment triples and compressing the triples in the fragment's target version. This suggests that while HDT favors efficient processing of triple pattern requests, using this encoding in the context of writable linked data has clear limitations. Additionally, our experimental setup with fragments of up to 10 million triples clearly exacerbates these limitations. Nevertheless, our experiments still show that it is possible to process queries over previous dataset versions within a relatively low query timeout, i.e., 31 out of 40 queries were executed within a timeout of 20 minutes; all timed out queries are either queries in the L and CH query groups, resulting in larger fragments to be materialized and therefore longer materialization time, or including triple patterns with variables as predicates, resulting in a large number of fragments to be materialized. Notice that these queries time out even when processed by state-of-the-art federated systems [17].

Update Overhead. To assess how consensual updates in ColChain scale in relation to the size of the community and the size of the fragments, we ran experiments where we applied updates to 338 randomly chosen fragments (out of the 4004 fragments in the network) of varying sizes in communities

of different sizes. We measured the update overhead time (UOT), including the time it takes for nodes within the community to forward acceptance messages. In this experiment, the updates are performed by a random non-owner node, and we assume that the participants in the community instantly accept the transactions. Figure D.7c shows the update time for different transactions applied to the 338 randomly chosen fragments. While the number of messages sent between nodes should rise polynomially with the community size (from 20 for the communities with five participants to 380 for communities with 20 participants), this has relatively little impact on the overhead of the updates. In line with the versioning experiments, the materialization of the updated file represents the biggest overhead. This highlights the limitations with updates that HDT entails.

7.4 Summary

Our experimental evaluations clearly show that ColChain is able to not only support consensual updates and query processing over previous dataset versions, but achieves comparable query processing performance to state-of-the-art decentralized architectures. The overhead of processing queries over previous dataset versions is mostly affected by the materialization of the older fragment itself, and ColChain is able to handle increasing chain lengths relatively efficiently in comparison. Furthermore, the overhead of consensual updates to fragments is also mostly affected by the materialization of the new version rather than the validation protocols. This highlights that the used RDF encoding has an impact on the performance with regards to versioning and consensual updates, and that a different encoding that allows for efficient updates could be more suitable for ColChain. Overall, our experimental evaluation clearly shows that ColChain is able to efficiently handle consensual updates, provides an approach to processing queries over previous versions that is scalable in relation to the length of the chain, and achieves good query processing performance in comparison to state-of-the-art decentralized architectures.

8 Conclusion

In this paper, we presented ColChain (COLlaborative knowledge CHAINs), a novel decentralized system that allows for users to provide updates to published knowledge graphs while enabling querying previous dataset versions. ColChain divides unstructured Peer-to-Peer networks into smaller communities of nodes and applies community-based chains of updates to data fragments. By relying on the consensus of participating nodes in a community, ColChain is able to let nodes propose and vote on updates to the

fragments. Furthermore, by structuring changes to fragments over time as chains, ColChain allows nodes to roll back updates and obtain query answers over previous dataset versions. Our experimental evaluation shows that ColChain achieves comparable performance to state-of-the-art interfaces, while the overhead of processing queries over previous dataset versions is mostly affected by the materialization of the older fragment and the used RDF encoding rather than the validation protocol. As part of our future work, we will investigate how to include measures to detect and avoid malicious activity. Furthermore, we will assess whether sharing the query processing load between nodes could improve query processing performance in ColChain and assess alternative RDF compression techniques to improve efficiency when processing queries over previous versions. We also plan to explore the possibility of using different consensus protocols that make active participation of users more scalable, e.g., by letting fragment owners specify a qualified majority or detecting malicious updates automatically. This could be done by expanding the metadata triples and adding routines for alternative consensus strategies.

References

[1] I. Abdelaziz, E. Mansour, M. Ouzzani, A. Aboulnaga, and P. Kalnis, "Lusail: A system for querying linked data at scale," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 485–498, 2017.

[2] M. Acosta, A. Zaveri, E. Simperl, D. Kontokostas, S. Auer, and J. Lehmann, "Crowdsourcing linked data quality assessment," in *ISWC 2013*, 2013, pp. 260–276.

[3] C. Aebeloe, I. Keles, G. Montoya, and K. Hose, "Star pattern fragments: Accessing knowledge graphs through star patterns," *CoRR*, vol. abs/2002.09172, 2020. [Online]. Available: https://arxiv.org/abs/2002.09172

[4] C. Aebeloe, G. Montoya, and K. Hose, "A decentralized architecture for sharing and querying semantic data," in *ESWC 2019*, 2019, pp. 3–18.

[5] ——, "Decentralized indexing over a network of rdf peers," in *ISWC 2019*, 2019, pp. 3–20.

[6] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "CAPER: A cross-application permissioned blockchain," *VLDB*, vol. 12, no. 11, pp. 1385–1398, 2019.

[7] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?" in *ISWC 2013*, 2013, pp. 277–293.

[8] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, and A. Polleres, "SMART-KG: hybrid shipping for SPARQL querying on the web," in *WWW 2020*, 2020, pp. 984–994.

[9] A. Azzam, C. A. I. Keles, G. Montoya, A. Polleres, and K. Hose, "WiseKG: balanced access to web knowledge graphs," in *WWW 2021*, 2021.

[10] M. Cai and M. R. Frank, "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *WWW*, 2004, pp. 650–657.

[11] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *ICDCS 2002*, 2002, pp. 23–32.

[12] M. Dumontier, A. Callahan, J. Cruz-Toledo, P. Ansell, V. Emonet, F. Belleau, and A. Droit, "Bio2rdf release 3: A larger, more connected network of linked data for the life sciences," in *ISWC 2014 Posters & Demonstrations Track*, vol. 1272, 2014, pp. 401–404.

[13] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "Blockchaindb - A shared database on blockchains," *PVLDB*, vol. 12, no. 11, pp. 1597–1609, 2019.

[14] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[15] D. Graux, G. Sejdiu, H. Jabeen, J. Lehmann, D. Sui, D. Muhs, and J. Pfeffer, "Profiting from kitties on ethereum: Leveraging blockchain RDF with SANSA," in *ISWC Posters and Demos*, 2018.

[16] O. Hartig and C. B. Aranda, "Bindings-restricted triple pattern fragments," in *OTM 2016 Conferences*, 2016, pp. 762–779.

[17] A. Hasnain, M. Saleem, A. N. Ngomo, and D. Rebholz-Schuhmann, "Extending largerdfbench for multi-source data at scale for SPARQL endpoint federation," in *SSWS@ISWC*, 2018, pp. 28–44.

[18] L. Heling, M. Acosta, M. Maleshkova, and Y. Sure-Vetter, "Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study," in *ISWC 2018*, 2018, pp. 86–102.

[19] L. D. Ibáñez, H. Skaf-Molli, P. Molli, and O. Corby, "Col-graph: Towards writable and scalable linked open data," in *ISWC 2014*, 2014, pp. 325–340.

[20] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi, "Atlas: Storing, updating and querying RDF(S) data on top of DHTs," *J. Web Sem.*, vol. 8, no. 4, 2010.

[21] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John, "UniStore: Querying a DHT-based universal storage," in *ICDE 2007*, 2007, pp. 1503–1504.

[22] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee, "A demonstration of the solid platform for social web applications," in *WWW 2016 Posters and Demos*, 2016, pp. 223–226.

[23] S. Metzger, K. Hose, and R. Schenkel, "Colledge: a vision of collaborative knowledge networks," in *SSW 2012*, 2012, pp. 1–8.

[24] T. Minier, H. Skaf-Molli, and P. Molli, "Sage: Web preemption for public SPARQL query services," in *WWW*, 2019, pp. 1268–1278.

[25] G. Montoya, C. Aebeloe, and K. Hose, "Towards efficient query processing over heterogeneous RDF interfaces," in *DeSemWeb@ISWC*, 2018.

[26] G. Montoya, H. Skaf-Molli, and K. Hose, "The Odyssey Approach for Optimizing Federated SPARQL Queries," in *ISWC 2017*, 2017, pp. 471–489.

[27] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[28] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, pp. 120–126, 1978.

[29] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo, "LSQ: the linked SPARQL queries dataset," in *ISWC 2015*, 2015, pp. 261–269.

[30] M. Saleem and A. N. Ngomo, "Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation," in *ESWC 2014*, 2014, pp. 176–191.

[31] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: A Federation Layer for Distributed Query Processing on Linked Open Data," in *ESWC 2011*, 2011, pp. 481–486.

[32] M. Sopek, P. Gradzki, W. Kosowski, D. Kuzinski, R. Trójczak, and R. Trypuz, "Graphchain: A distributed database with explicit semantics and chained RDF graphs," in *WWW Companion*, 2018, pp. 1171–1178.

[33] C. Stadler, J. Lehmann, K. Höffner, and S. Auer, "Linkedgeodata: A core for a web of spatial open data," *Semantic Web*, vol. 3, no. 4, pp. 333–354, 2012.

[34] P. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan, and C. B. Aranda, "SPARQLES: monitoring public SPARQL endpoints," *Semantic Web*, vol. 8, no. 6, pp. 1049–1065, 2017.

[35] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: A low-cost knowledge graph interface for the Web," *J. Web Sem.*, vol. 37-38, pp. 184–206, 2016.

[36] D. Vrandecic and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Commun. ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[37] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: a survey," *IJWGS*, vol. 14, no. 4, pp. 352–375, 2018.

Paper E

A Demonstration of ColChain: Collaborative Knowledge Chains

Christian Aebeloe, Gabriela Montoya, Katja Hose

The paper has been published in the
Proceedings of the ISWC 2021 Posters, Demos and Industry Tracks (ISWC 2021),
2021.

Abstract

The current architecture of the Semantic Web fully relies on the individual data providers to maintain access to their data and to keep their data up to date. While this may seem like a practical and straightforward solution, it often results in the data being unavailable or outdated. In this paper, we present a fully functioning client along with a user-friendly interface for ColChain*, a system that increases availability of knowledge graphs and enables users to update the data in a community-driven way while still allowing them to query old versions.*

Aebeloe C., Montoya G., Hose K. (2021) A Demonstration of ColChain: Collaborative Knowledge Chains. In: *Proceedings of the ISWC 2021 Posters, Demos and Industry Tracks (ISWC 2021)*.
The layout has been revised.

1 Introduction

In recent years, the continued development of Semantic Web technologies has led to a rapid increase of the amount of data published as Linked Open Data. However, the proliferation of data on the Semantic Web and the fact that we currently rely on the data providers to maintain access to their datasets and keep them up-to-date represents a significant burden on the data providers [1, 5]. As a result, public SPARQL endpoints often experience downtime [3] and available data is sometimes outdated [2].

```
SELECT ?pr2 WHERE {
    dbr:President_of_the_United_States dbo:incumbent ?pr1 .
    ?pr1 dbo:party ?pa .
    ?pr2 dct:subject dbr:Category:Presidents_of_the_United_States .
    ?pr2 dbo:party ?pa
}
```

Listing E.1: SPARQL query Q that finds former U.S. presidents of the same party as the current (incumbent) U.S. president.

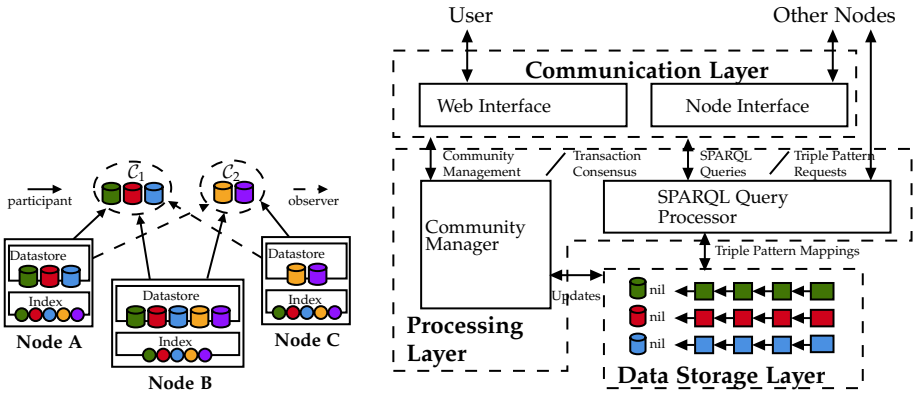
Consider, for instance, the query Q in Listing E.1. Q finds all former U.S. presidents that have been a member of the same party as the current (incumbent) U.S. president. However, as of the submission of this paper, processing Q over the latest DBpedia release (version 2021-01)[1] results in `?pr1`, i.e., the current (incumbent) president, being bound to `dbr:Donald_Trump` although the inauguration of President Biden took place months ago. While this is likely to be changed in the next release, the delay in the update shows that information available on the Semantic Web is not always up-to-date.

In this paper, we demonstrate ColChain (COLlaborative knowledge CHAINs) [2], a system that builds on unstructured Peer-to-Peer (P2P) networks [1] and uses replication of data fragments across several nodes to maintain high availability. Furthermore, ColChain enables users to collaboratively update datasets while also allowing users to process queries over previous versions. ColChain divides the P2P network into smaller *communities* of nodes that collaborate on keeping certain data up-to-date relying on community-wide consensus. Updates in ColChain are stored in blockchain-like chains; when a consensus for an update is reached, it is applied to the end of the chain. This allows any user to propose updates to any dataset while making malicious updates less likely. Furthermore, the update chains allow users to access previous versions of the datasets. While [2] presents the theoretical framework, this demo paper presents a working ColChain implementation with a user-friendly interface[2].

This paper is structured as follows. In Section 2, we present an architectural overview of ColChain while in Section 3 we describe the demonstration that will be conducted at the conference.

[1] https://www.dbpedia.org/resources/latest-core/

[2] The client and source code are available at https://relweb.cs.aau.dk/colchain/



(a) Example ColChain network. **(b)** Architecture of a ColChain node.

Fig. E.1: An example ColChain network and the architecture of a ColChain node (adapted from [2]).

2 System Overview

Figure E.1a shows an example of a ColChain network that consists of three nodes that store data from two communities, where the nodes either *participate* in or *observe* the communities. Participanting nodes share a set of data fragments with the community they participate in and collaborate on keeping those fragments up-to-date. In Figure E.1a, since node A participates in community $\mathcal{C}_1$, it stores the fragments in its local datastore. Furthermore, A observes community $\mathcal{C}_2$, and thus only indexes the fragments, relying on asking either node B or C (i.e., participants) to access the data. Due to space restrictions, we do not go into details with aspects, such as how to create and maintain communities, but refer the interested reader to [2] for a more technically detailed description.

2.1 Architecture of a ColChain Client

A ColChain node generally consists of several architectural layers as illustrated in Figure E.1b. These layers are as follows.

Communication Layer. The communication layer exposes two components: the Web interface and the node interface. The Web interface provides a GUI that allows users to interact with the system, e.g., to issue SPARQL queries (on current or previous versions of the data), propose updates, and decide whether to accept or reject updates proposed by other users. The node interface accepts messages from other nodes, e.g., when another participant accepts an update.

Processing Layer. The processing layer consists of two components: the

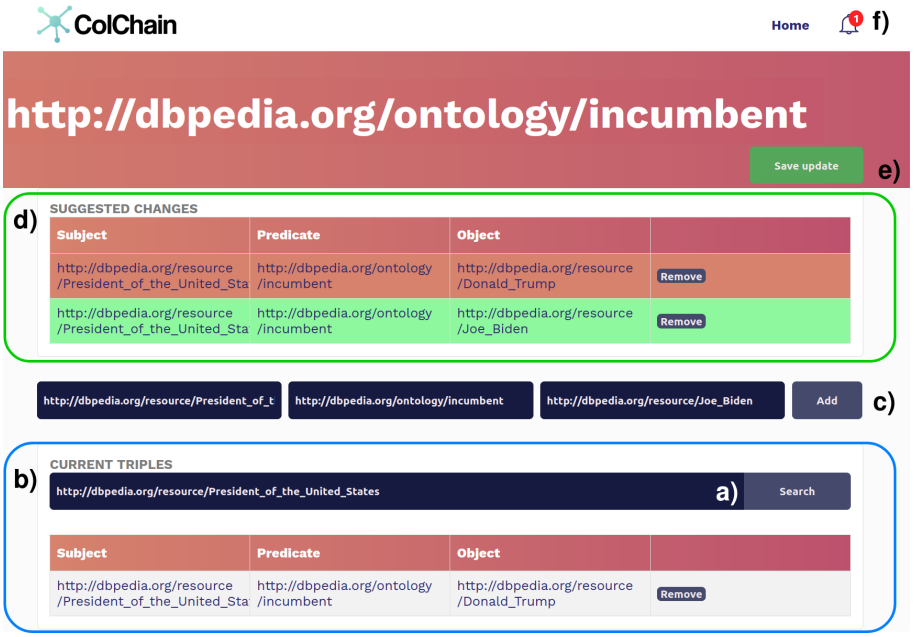


Fig. E.2: ColChain's graphical interface when proposing an update to a fragment.

community manager and the query processor. The community manager validates updates, and manages chains and fragments as well as community membership. The query processor is able to process SPARQL queries over current and old dataset versions available at any user-specified point in time.

Data Storage Layer. The data storage layer contains the node's local data store. ColChain nodes use HDT [4] as backend for storing data fragments. Changes to fragments are applied to the data storage layer by the community manager and appended to the chain for the given fragment.

2.2 Graphical User Interface for ColChain

Consider again query Q from Listing E.1 and a user who wants to suggest an update to obtain the expected result. Figure E.2 shows how the user interacts with ColChain to propose an update over a fragment. The user searches for the URI `dbr:President_of_the_United_States` (Figure E.2a) and finds the triple with `dbr:Donald_Trump` as object (Figure E.2b), which they then remove. The user then adds the triple with `dbr:Joe_Biden` as object to the fragment (Figure E.2c). Figure E.2d shows the changes made by the user. Once the user saves the update (Figure E.2e), it is forwarded to the other participants in the community, which are notified (Figure E.2f). Once a majority accepts the update, it is applied across the community.

3 Demonstration

At the conference, we will demonstrate ColChain using two separate scenarios that attendees can explore. We will run a network with the data from LargeRDFBench [6], which comprises 13 interlinked datasets with over a billion triples in total. Furthermore, we will run a separate network with data from a subset of DBpedia that includes update chains back to version 2015-04, i.e., attendees will have the opportunity to explore query answers over different versions of DBpedia. ColChain will be showcased using networks with varying numbers of nodes and community sizes that follow different distributions (e.g., Zipfian as in [2]). A video demonstration of ColChain using the DBpedia scenario is available on our website[3].

To ease interaction with the system, we will provide several interesting SPARQL queries for attendees to explore each scenario. Attendees will be invited to build upon these queries, formulate queries on their own, explore query answers over different versions, and propose updates. For instance, attendees could propose the update shown in Figure E.2. Query Q from Listing E.1 could then be processed over the updated data as well as over DBpedia version 2015-04 when `?pr1` would be bound to `dbr:Barack_Obama`.

References

[1] C. Aebeloe, G. Montoya, and K. Hose, "A decentralized architecture for sharing and querying semantic data," in *ESWC 2019*, 2019, pp. 3–18.

[2] ——, "ColChain: Collaborative linked data networks," in *WWW 2021*, 2021, pp. 1385–1396.

[3] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?" in *ISWC 2013*, 2013, pp. 277–293.

[4] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[5] L. Heling, M. Acosta, M. Maleshkova, and Y. Sure-Vetter, "Querying Large Knowledge Graphs over Triple Pattern Fragments: An Empirical Study," in *ISWC 2018*, 2018, pp. 86–102.

[6] M. Saleem, A. Hasnain, and A. N. Ngomo, "Largerdfbench: A billion triples benchmark for SPARQL endpoint federation," vol. 48, 2018, pp. 85–125.

[3] `https://relweb.cs.aau.dk/colchain#demonstration`

Paper F

Optimizing SPARQL Queries over Decentralized Knowledge Graphs

Christian Aebeloe, Gabriela Montoya, Katja Hose

Unpublished Manuscript

Abstract

While the Web of Data in principle offers access to a wide range of interlinked data, the architecture of the Semantic Web today relies mostly on the data providers to maintain access to their data through SPARQL endpoints. Several studies, however, have shown that such endpoints often experience downtime, meaning that the data they maintain becomes inaccessible. While decentralized systems based on Peer-to-Peer (P2P) technology have previously shown to increase the availability of knowledge graphs, even when a large proportion of the nodes fail, processing queries in such a setup can be an expensive task since data necessary to answer a single query might be distributed over multiple nodes. In this paper, we therefore propose an approach to optimizing SPARQL queries over decentralized knowledge graphs, called Lothbrok*. While there are potentially many aspects to consider when optimizing such queries, we focus on three aspects: cardinality estimation, locality awareness, and data fragmentation. We empirically show that* Lothbrok *is able to achieve significantly faster query processing performance compared to the state of the art when processing challenging queries as well as when the network is under high load.*

Unpublished manuscript.
The layout has been revised.

1 Introduction

Due to the popularity of decentralized knowledge graphs on the Web, more and increasingly large knowledge graphs encoded in RDF are becoming available [36]. Furthermore, RDF knowledge graphs made available today are becoming exceedingly large. For instance, Wikidata [68] and Bio2RDF [21] contain more than 14 billion triples each. As a result, data providers experience an increasing burden of maintaining access to the datasets; and without any monetary incentives to do so, datasets often end up becoming unavailable [4, 12, 65] and outdated [6].

In recent years, several decentralized systems [3, 4, 6, 13, 43, 66] have been proposed to alleviate the aforementioned burden from the data providers by reducing the computational load required to keep the data available, albeit using different methods to do so. For instance, Linked Data Fragments (LDF)-based approaches [3, 13, 14, 33, 66] reduce the computational load on the server by distributing some of the query processing effort to the client, ensuring that the server only processes requests with low time complexity. On the other hand, Peer-to-Peer (P2P) systems [4, 6, 43] remove the centralized point of failure that a server represents and replicate the data across several nodes in a decentralized fashion, ensuring that even if the uploading node fails, the data is still accessible. For instance, RDFPeers [17] uses a structured overlay over a P2P network that relies on Dynamic Hash Tables (DHTs) to determine where to replicate certain data. However, in situations where nodes frequently leave or join the network (i.e., churn), and data is often uploaded to the network, nodes have to go through a costly adjustment process to update the overlay and redistribute the data. Instead, systems like Piqnic [4] and ColChain [6] use unstructured P2P systems as foundation, where there is no global control over where data is replicated, making the network more stable under churn.

ColChain builds upon Piqnic and divides the entire network into communities of nodes that not only replicate the same data, but also collaborate on keeping certain data (fragments) up-to-date. This is done by using blockchain technology [26, 52, 63, 69] where *chains* of updates maintain the history of changes to the data fragments. By linking such update chains to the data fragments in a community, ColChain allows community participants to collaborate on keeping the data up-to-date while using consensus to make malicious updates less likely and allowing users to roll-back updates to an earlier version on request. Furthermore, the decentralized nature of ColChain also increases the availability of the uploaded data by replicating the data on nodes within the community.

Nevertheless, while Piqnic and ColChain already use decentralized indexes [5] to determine where data is located during query time, subgraphs

needed to answer a query are usually scattered across multiple nodes. Furthermore, the indexes provide limited information that prevents the nodes from considering locality and accurately estimating join cardinalities when optimizing queries. As a result, such systems often experience an unnecessarily large amount of intermediate results when processing a query. This problem is exacerbated by the decentralized nature of the systems, since the intermediate results have to be transferred between nodes, causing a significant communication overhead.

While there are potentially many aspects to consider when optimizing queries in a decentralized setup, we will focus on three such aspects: cardinality estimation, locality awareness, and data fragmentation. Suboptimal solutions to any of these three aspects can lead to an increased communication overhead and lower performance. For instance, while fragmenting large knowledge graphs into smaller fragments ensures that nodes do not have to replicate entire knowledge graphs, using a fragmentation technique that spreads out the data relevant to a single (sub)query across several fragments can increase the communication overhead since nodes might have to send an excessive number of requests to obtain all relevant data to answer a particular query [7, 8, 23, 38]. On the other hand, inaccurate cardinality estimations can lead to a suboptimal join strategy that increases the amount of intermediate results and therefore runtime [50, 53]. And while several approaches have proposed reasonably accurate cardinality estimation techniques [50, 53, 55] over knowledge graphs, and for federated engines in particular [29, 37, 50, 64], such approaches cannot easily be transferred to a decentralized setup since nodes in a decentralized setup lack a global overview of the network and the data is scattered across multiple nodes. Finally, considering locality of the data when processing queries can help ensure that larger subqueries are delegated to nodes that can process them without communicating with other nodes, lowering the data transfer overall.

Nevertheless, while an optimization approach that maximizes the degree to which entire queries can be processed by a single node could decrease the communication overhead, a study [7] found that processing entire queries on one node can actually decrease the overall performance when the network is under heavy load, and that it is equally important to balance out the query load between nodes. As such, there is a need for a more holistic approach to query optimization that is able to delegate the processing of subqueries to other nodes in the network, thus reducing the communication overhead to the extent possible. For instance, query optimization techniques that are based on star-shaped subqueries have previously been shown to increase performance by at least an order of magnitude [3, 13, 14, 67]. This, and the fact that conjunctive subqueries are relatively efficient to process [56], means that decomposing and processing queries based on star-shaped subqueries can significantly reduce the communication overhead in decentralized systems.

In this paper, we therefore extend our work on PIQNIC [4] and COLCHAIN [6] in three aspects that work together to reduce the communication overhead when processing SPARQL queries, and in doing so, improve query processing performance in an approach that we call LOTHBROK. LOTHBROK adapts Characteristic Sets [3, 13, 14, 53] to fragment data in decentralized P2P systems. Furthermore, LOTHBROK builds upon Prefix-Partitioned Bloom Filters (PPBFs) [5] and proposes a new indexing scheme called Semantically Partitioned Bloom Filters (SPBFs) to obtain more accurate cardinality estimations. Lastly, LOTHBROK also introduces a locality-aware query optimization strategy that takes advantage of the SPBF indexes and is able to delegate the processing of (sub)queries to neighboring nodes in the network holding relevant data. We evaluate LOTHBROK thoroughly using LargeRDFBench [59], a benchmark suite for federated RDF systems that comprises 13 datasets with over a billion triples and includes 40 queries of varying complexity and sizes of intermediate results. Furthermore, we evaluate LOTHBROK using synthetic data and queries from WatDiv [9] to test the scalability of LOTHBROK under load. In summary, we make the following contributions:

- A data fragmentation technique that builds on Characteristic Sets [53]
- SPBF indexes adapted to the characteristic set fragmentation technique
- A cardinality estimation approach over decentralized RDF fragments using the SPBF indexes to provide more accurate cardinality estimations
- A locality-aware query optimization algorithm that uses SPBF indexes to delegate subqueries to neighboring nodes and reduce the communication overhead
- A thorough experimental evaluation of the impact of the presented techniques on query processing performance using real-world data from a well-known benchmark suite, and large-scale synthetic datasets

The paper is structured as follows: Section 2 discusses related work while Section 3 describes background information. Then, Section 4 presents LOTHBROK, Section 5 details how LOTHBROK optimizes queries, and Section 6 describes the query execution approach, while Section 7 presents our experimental evaluation. Lastly, Section 8 concludes the paper with an outlook to future work.

2 Related Work

The availability problem has prompted significant amount of research in the areas of decentralized query processing and decentralized architectures for knowledge graphs. In this section, we therefore discuss existing approaches

related to Lothbrok; client-server architectures, federated systems, and P2P systems.

2.1 Client-Server Architectures

SPARQL endpoints are Web services providing an HTTP interface that accepts SPARQL queries and remain some of the most popular interfaces for querying RDF data on the Web. However, several studies [12, 65] have found that such endpoints are often unavailable and experience downtime.

Linked Data Fragment (LDF) interfaces, such as Triple Pattern Fragments (TPF) [66], attempt to increase the availability of the server by shifting some of the query processing load towards the client while the server only processes requests with low time complexity. For instance, TPF servers only process individual triple patterns while the TPF clients process joins and other expensive operations. Today, several TPF clients exist that rely on either a greedy algorithm [66], a metadata based strategy [35], or star-shaped query decomposition combined with adaptive query processing techniques [1] to determine the join order of the triple patterns in a query. However, while in all these approaches the server can handle more concurrent requests in comparison to SPARQL endpoints without becoming unresponsive, TPF naturally incurs a large network overhead when processing queries since intermediate bindings from previously evaluated triple patterns are transferred along with subsequently evaluated triple patterns to limit the amount of intermediate results, one by one. Furthermore, studies found that the performance of TPF is heavily affected by the type of triple pattern (i.e., the position of variables in the triple pattern) [33] and the shape of the query [48, 49].

Several different systems have since been proposed to lower the network overhead. For instance, Bindings-Restricted TPF (brTPF) [30] bulks bindings from previously evaluated triple patterns such that multiple bindings can be attached to a single request. While this reduces the number of requests made for a triple pattern, it still incurs a somewhat large data transfer overhead, since each request still evaluates a single triple pattern. hybridSE [47] combines a brTPF server with a SPARQL endpoint and takes advantage of the strengths of each approach; subqueries with large numbers of intermediate results are sent to the SPARQL endpoint to overcome the limitations posed by LDF systems. However, hybridSE often answers complex queries using the SPARQL endpoint and is thus vulnerable to server failure.

To further limit the network overhead, Star Pattern Fragments (SPF) [3] clients send conjunctive subqueries in the shape of stars (star patterns) to the server and process more complex patterns locally on the client. Such conjunctive subqueries can be processed relatively efficiently by the server [56], which results in the transfer of significantly fewer intermediate results than in systems like TPF and brTPF. On the other hand, Smart-KG [14] ships

predicate-family partitions (i.e., characteristic sets) to the client and processes the entire query locally; however, triple patterns with infrequent predicate values (according to a certain threshold) are sent to and evaluated by the server. While this takes advantage of the distributed resources that the clients possess, Smart-KG often ends up transferring excessive amounts of data unnecessarily since entire partitions of a dataset are transferred regardless of any bindings from previously evaluated star patterns. WiseKG [13] combines SPF and Smart-KG and uses a cost model to determine which strategy (SPF or Smart-KG) is the most cost-effective to process a given star-shaped subquery. Like SPF and Smart-KG, WiseKG processes more complex patterns on the client. Nevertheless, all the aforementioned LDF approaches rely on a centralized server or a fixed set of servers that are subject to failure.

Lastly, different from LDF approaches, SaGe [46] decreases the load on the server by suspending queries after a fixed time quantum to prevent long-running queries from exhausting server resources; the queries can then be restarted by making a new request to the server. However, SaGe processes entire, and possibly complex, queries on the server, and as stated above, such servers are subject to failure.

2.2 Federated Systems

Federated systems enable answering queries over data spread out across multiple independent SPARQL endpoints [2, 18, 25, 39, 62] or LDF servers [32] offering access to different datasets. While such approaches spread out query processing over several servers, lowering the load on each individual server, they sometimes generate suboptimal query execution plans that increase the number of intermediate results and the load on individual servers [41]. As such, several approaches [29, 37, 50, 51, 60, 64] have attempted to optimize federated queries in different ways. For instance, [62] builds an index over time by remembering which endpoints in the federation can provide answers to which triple patterns. Furthermore, [51] decomposes queries into subqueries that can be evaluated by a single endpoint. While [51] uses a similar query decomposition strategy as Lothbrok, they target federations over SPARQL endpoints, and as previously mentioned, such endpoints suffer from availability issues. On the other hand, [50, 60] estimate the selectivity of joins to produce more efficient join plans. For instance, [50] uses characteristic sets [53] and pairs [27] to index the data in the federation and combines this with Dynamic Programming (DP) to optimize query execution plans. Furthermore, [32] proposes an interface for processing federated queries over heterogeneous LDF interfaces. To achieve this, the query optimizer is adapted to the characteristics of the different interfaces as well as the locality of the data, i.e., knowledge of which nodes hold which data. Inspired by these approaches, Lothbrok fragments knowledge graphs based on char-

acteristic sets and uses a similar cardinality estimation technique to optimize join plans in consideration of data locality in the network.

2.3 Peer-to-Peer Systems

Peer-to-Peer (P2P) systems [4, 6, 17, 23, 42, 43, 45] tackle the availability issue from a different perspective: by removing the central point of failure completely and replicating the data across multiple nodes in a P2P network, they can ensure the data remains available even if the original node that uploaded the data fails. As such, they consist of a set of nodes (often resource limited) that act both as servers and clients, maintaining a limited local datastore. The structure of the network, i.e., connections between the nodes, as well as data placement (data allocation), varies from system to system. For instance, some systems [17, 42, 43] enforce data placement by applying a structured overlay over the network, such as Dynamic Hash Tables (DHTs) [44]. On the other hand, PIQNIC [4] imposes no structure on top the network; nodes are connected randomly to a set of neighbors that are shuffled periodically with another node's neighbors to increase the degree of joinability between the fragments of neighboring nodes. Lastly, COLCHAIN [6] extends PIQNIC and divides the entire network into smaller communities of nodes that collaborate on keeping certain data available and up-to-date. By applying community-based ledgers of updates and relying on a consensus protocol within a community, COLCHAIN lets users actively participate in keeping the data up-to-date.

Each P2P system has different ways of processing queries. For instance, due to the lack of global knowledge over the network, basic P2P systems have to flood the network with requests for a given horizon to increase the likelyhood of receiving complete query results. To counteract this, distributed indexes [5, 20, 64] like Prefix-Partitioned Bloom Filter (PPBF) indexes [5] determine which nodes may include relevant data for a given query and thus allow the system to prune nodes from consideration during query optimization. Yet, the aforementioned systems still experience a significant overhead partly caused by inaccurate cardinality estimations, query optimization that does not consider the locality of data, as well as data fragmentation that splits up closely related data. For instance, PIQNIC and COLCHAIN both use a predicate-based fragmentation strategy that creates a fragment for each predicate. This, together with the replication and allocation strategy used, means that data relevant to a single query is distributed over a significant number of fragments and nodes.

However, while an approach that maximizes the degree to which entire queries can be processed by one node can lower the communication overhead, distributing some of the query processing load across multiple nodes is equally important when optimizing queries in a decentralized context [7]

to avoid overloading individual nodes. As such, LOTHBROK limits the communication overhead by fragmenting data based on characteristic sets and introducing a new indexing scheme that lets nodes take advantage of the fragmentation to more accurately estimate subquery cardinality and distribute the processing of subqueries to nodes in the network based on data locality. Furthermore, since fragments are created based on characteristic sets, entire star patterns can be processed efficiently by single nodes, further distributing the query processing load, lowering the communication overhead at the same time, and increasing the query throughput.

3 Background

A commonly used format for storing semantic data is the Resource Description Framework (RDF) [16]. RDF structures data as triples, defined as follows.

Definition F.1 (RDF Triple)
Let I, B, and L be the disjoint sets of IRIs, blank nodes, and literals. An RDF triple is a triple t of the form $t = (s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where s, p, and o are called subject, predicate, and object.

Given the definition of an RDF triple, a *knowledge graph* $\mathcal{G}$ is a finite set of RDF triples. The most popular language to query knowledge graphs is SPARQL [19]. A SPARQL query consists of one or more *triple patterns*. A triple pattern t is a triple of the form $t = (s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ where V is the set of all variables. A Basic Graph Pattern (BGP) is a set of conjunctive triple patterns. Without loss of generality, we focus our discussion in the main part of this paper on BGPs and describe in Section 5 how our approach can support other operators, such as `UNION` and `OPTIONAL`; our experimental evaluation in Section 7 includes queries with a variety of SPARQL operators including `UNION` and `OPTIONAL`.

A complex BGP P can be decomposed into a set of *star patterns*. A star pattern P' is a set of triple patterns that share the same subject, i.e., $\forall t_1 = (s_1, p_1, o_1), t_2 = (s_2, p_2, o_2)$ such that $t_1, t_2 \in P'$, it is the case that $s_1 = s_2$. Note that while star patterns can be defined as both subject-based and object-based star patterns, for ease of presentation, we focus on subject-based star patterns only since subject-subject joins are much more common in real query loads [61]; LOTHBROK can trivially be adapted to object-based star patterns by using the same principles presented in this paper for object-object joins rather than subject-subject joins.

Definition F.2 (Star Decomposition [3])
Given a BGP $P = \{t_1, \ldots, t_n\}$ with subjects $S_P = \{s_1, \ldots, s_m\}$, the *star decomposition* of P, $\mathcal{S}(P) = \{P_s(P) \mid s \in S_P\}$, is a set of star patterns $P_s(P)$ for each

$s \in S_P$, such that $P = \cup_{s \in S_p} P_s(P)$ where $P_s(P) = \{(s', p', o') \mid (s', p', o') \in P \wedge s' = s\}$.

The answer to a BGP P over a knowledge graph $\mathcal{G}$ is a set of *solution mappings*, defined as follows.

Definition F.3 (Solution mapping [6, 66])
Given a BGP P and a knowledge graph $\mathcal{G}$, the sets $I_\mathcal{G}$, $B_\mathcal{G}$, and $L_\mathcal{G}$ are the sets of IRIs, blank nodes, and literals in $\mathcal{G}$, and V_P is the set of variables in P, a *solution mapping* μ is a partial mapping $\mu : V_P \mapsto (U_\mathcal{G} \cup B_\mathcal{G} \cup L_\mathcal{G})$.

Given a BGP P and a solution mapping μ, the notation $\mu[P]$ denotes the triple (patterns) obtained by replacing variables in P according to the bindings in μ. Furthermore, given a knowledge graph $\mathcal{G}$ and BGP P, $[[P]]_\mathcal{G}$ denotes the set of solution mappings that constitute the answer to P over $\mathcal{G}$, i.e., $\forall \mu \in [[P]]_\mathcal{G}$, $\mu[P] \in \mathcal{G}$, and $\forall T \in \mu[P]$, T is a set of *matching* triples to P, denoted $T[P]$. Furthermore, $dom(\mu)$ returns the *domain* of μ, i.e., the set of variables that are bound in μ and $vars(P)$ returns the variables in P.

3.1 Peer-to-Peer

In its simplest form, an unstructured P2P system consists of a set of interconnected nodes that all maintain a local datastore managing a set of (partial) knowledge graphs, where each node maintains a local view over the network, i.e., a set of *neighboring* nodes (nodes within the local view over the network).

ormally, we define a P2P network N as a set of interconnected nodes $N = \{n_1, \ldots, n_n\}$ where each node maintains a local datastore and a local view over the network. The data uploaded to a node in N is replicated throughout the network. Furthermore, in line with previous work [5, 6], each node maintains a distributed index describing the knowledge graphs reachable within a certain number of steps (also known as hops), called the *horizon* of a node. A node n is defined as follows:

Definition F.4 (Node [4, 5])
A node n is a triple $n = (G, I, N_n)$ where:

- G is the set of knowledge graphs in n's local datastore
- I is n's distributed index
- N_n is a set of neighboring nodes

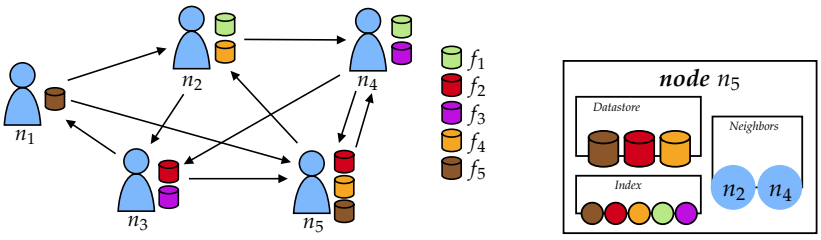
While maintaining the structure of the network is important for P2P systems, it is not relevant for the data and query processing techniques that this paper is focusing on. As such, we do not go into detail on network topology, data replication and allocation, and periodic shuffles. Instead, we refer the interested reader to related work such as [4, 6] for more details. In the following, we define data fragmentation and introduce a running example.

In line with previous work [4, 6], and to avoid having to replicate large knowledge graphs throughout the network, LOTHBROK divides knowledge graphs into smaller disjoint *fragments*, i.e., partial knowledge graphs, which can be replicated more easily. Fragments can be obtained using a *fragmentation* function. A fragmentation function is a function that, given a knowledge graph, returns a set of disjoint fragments, and is formally defined as follows:

Definition F.5 (Fragmentation Function [4, 6])
A *fragmentation* function $\mathcal{F}$ is a function that maps a knowledge graph $\mathcal{G}$ to a set of *knowledge graph fragments*, i.e., $\mathcal{F} : \mathcal{G} \mapsto 2^{\mathcal{G}}$.

Different fragmentation functions can have different granularities. For instance, the most coarse-granular fragmentation function is $\mathcal{F}_C(\mathcal{G}) = \{\mathcal{G}\}$, i.e., the fragmentation function does not split up the original knowledge graph. COLCHAIN [6] as well as PIQNIC [4] use a *predicate-based* fragmentation function for $\mathcal{G}$, i.e., $\mathcal{F}_P(\mathcal{G}) = \{\{(s', p', o') \mid (s', p', o') \in \mathcal{G} \wedge p' = p\} \mid \exists s, o : (s, p, o) \in \mathcal{G}\}$, which creates a fragment for each unique predicate in $\mathcal{G}$. LOTHBROK uses a fragmentation function based on characteristic sets [53] (i.e., predicate families) that is detailed in Section 4.2.

The fragments created by the fragmentation function are replicated and allocated at multiple nodes in the network to ensure availability in case the original provider of the knowledge graph becomes unavailable and to enable load balancing. The replication and allocation factor are parameters of the underlying network; for instance, in PIQNIC [4], fragments are replicated and allocated across the node's neighbors, and nodes index all fragments available within a certain horizon. On the other hand, COLCHAIN [6] replicates and allocates fragments at nodes that participate within the same communities. Since this paper focuses on data fragmentation and query optimization, we omit details on data replication and allocation and refer the interested reader to related work [4, 6] for details.



(a) Example network N **(b)** Node n_5

Fig. F.1: (a) Example of an unstructured P2P network $N = \{n_1, \ldots, n_5\}$ and (b) architecture of a single node n_5 that indexes data within a horizon of 2 nodes.

Consider, as a running example, the unstructured P2P network in Figure F.1a consisting of five nodes ($N = \{n_1, \ldots, n_5\}$) that replicate a total of

five fragments ($f_1, \ldots, f_5$). In this example, each node maintains a set of two neighbors and each fragment is replicated across two nodes. For instance, node n_5 has $\{n_2, n_4\}$ as its set of neighbors, and replicates the fragments $\{f_2, f_4, f_5\}$ in its local datastore. While the running example is based on an unstructured network, such as the one presented in [4], Lothbrok could be adapted to more structured setups, such as the one presented in [6].

3.2 Distributed Indexes

To speed up query processing performance, systems like Piqnic [4] and ColChain [6] use distributed indexes [5, 20] to efficiently identify nodes holding relevant data for a given SPARQL query. The indexes capture information about the fragments stored locally at the node itself as well as information about fragments that can be accessed via its neighbors.

In [5, 6], a distributed index is formally defined as consisting of two mappings; (1) from a triple pattern to the set of fragments containing relevant data to the triple pattern, and (2) from a fragment to the set of nodes that store the fragment. Furthermore, to build the indexes for a node's local view over the network, nodes share partial indexes, i.e., partial mappings, for the fragments that they have access to, called index *slices*. In line with [5, 6], we define distributed indexes and index slices in the following.

Definition F.6 (Distributed Index [5, 6])
Let $\mathcal{N}$ be the set of nodes within a network, n be a node such that $n \in \mathcal{N}$, $\mathcal{T}$ be the set of all possible triple patterns, and $\mathcal{F}$ be the set of fragments that n has access to within its local view over the network. A *distributed index* on n is a tuple $I_n = (\nu, \eta)$ with $\nu : \mathcal{T} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^{\mathcal{N}}$. For a triple pattern t, $\nu(t)$ returns the set of fragments in $\mathcal{F}$ that t matches. For a fragment $f \in \mathcal{F}$, $\eta(f)$ returns the nodes on which f is located.

Given a node n, n's distributed index is denoted I_n. Given the definition of a distributed index, we define a *node mapping* as a mapping from a triple pattern t in a BGP P to a set of nodes that contain relevant fragments to t, as follows:

Definition F.7 (Node Mapping [5, 6])
For any BPG P and distributed index I, there exists a function $match(P, I)$ that returns a *node mapping* $M : P \mapsto 2^{\mathcal{N}}$, such that $\forall t \in P$, $M(t)$ returns the indexed nodes that have fragments holding data matching the triple t.

An index slice for a fragment is a partial mapping from triple patterns to the fragments that contain relevant triples to the triple patterns, as well as a mapping from the fragment to the nodes that replicate it, and is defined as follows:

Definition F.8 (Index Slice [5, 6])
Let f be a fragment. The index *slice* of f, s_f, is a tuple $s_f = (\nu', \eta')$, where $\nu'(t)$ returns $\{f\}$ if there exists a triple in f that matches t, and $\eta'(f)$ returns the set of all nodes that contain f in their local datastore. The function $s(f)$ returns the index slice describing f.

Index slices for the fragments that a node has access to are combined into a distributed index for that particular node using the $\oplus$ operator[1]. The distributed index is then used to check the relevancy and overlap of fragments during query time to optimize the query. Given a set of slices S, the index obtained by combining the slices in S, $I(S)$, can be computed using the formula in Equation F.1 [5, 6].

$$I(S) = \left(\bigoplus_{s \in S} s.\nu', \bigoplus_{s \in S} s.\eta' \right) \tag{F.1}$$

While the definition of distributed indexes allows for several different types of indexes, the index slices used in Pqnic [4] and ColChain [6] correspond to Prefix-Partitioned Bloom Filters (PPBFs) [5], which extend regular Bloom filters [15]. A Bloom filter $\mathcal{B}$ for a set S of IRIs such that $|S| = n$ is a tuple $\mathcal{B} = (\hat{b}, H)$ where $\hat{b}$ is a bitvector of size m and H is a set of k hash functions [5]. Each hash function in H maps the elements from S (i.e., IRIs) to a position in $\hat{b}$; these positions are thus set to 1 whereas the positions not mapped to by a function in H are 0. In other words, [5] represents the combined set of subjects and objects in a fragment in a prefix-partitioned bitvector. Looking up whether an element e is in S using the Bloom filter for S is done by hashing e using the hash functions in H and checking the value of each position in $\hat{b}$. If at least one of those positions is set to 0, it is certain that $e \notin S$. However, if all corresponding bits are set to 1, it is not certain that $e \in S$, since it could be a false positive caused by hash collisions, i.e., different values are mapped to the same positions in the underlying bitvector. In this case, we say that e *may* be in S, denoted $e \sqsubset S$.

To check the compatibility of two fragments relevant for conjunctive triple patterns, we check whether or not they produce any join results. To do this, we could check whether or not the intersection of the bitvectors describing the subjects and objects of the fragments is empty (i.e., if they have some IRI in common). Given two Bloom filters $\mathcal{B}_1 = (\hat{b}_1, H)$ and $\mathcal{B}_2 = (\hat{b}_2, H)$, the intersection of $\mathcal{B}_1$ and $\mathcal{B}_2$ is approximated by the logic `AND` operation between $\hat{b}_1$ and $\hat{b}_2$, $\mathcal{B}_1 \cap \mathcal{B}_2 \approx \hat{b}_1 \& \hat{b}_2$.

To avoid exceedingly large bitvectors, PPBFs partition the bitvector based on the prefix of the IRIs. A PPBF is formally defined in [5] as follows.

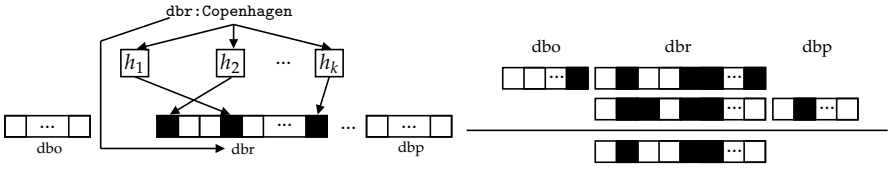
[1] $\oplus$ is defined in [5, 6] as $(f \oplus g)(x) = f(x) \cup g(x)$ if f and g are defined at x; $(f \oplus g)(x) = f(x)$ if f is defined at x; $(f \oplus g)(x) = g(x)$ if g is defined at x.

Definition F.9 (Prefix-Partitioned Bloom Filter [5])
A PPBF $\mathcal{B}^P$ is a 4-tuple $\mathcal{B}^P = (P, \hat{B}, \theta, H)$ where

- P a set of prefixes
- $\hat{B}$ is a set of bitvectors such that $\forall \hat{b}_1, \hat{b}_2 \in \hat{B} : |\hat{b}_1| = |\hat{b}_2|$
- $\theta : P \rightarrow \hat{B}$ is a prefix-mapping function
- H is a set of hash functions

For each $p_i \in P$, $\mathcal{B}_i = (\theta(p_i), H)$ is the Bloom Filter that encodes the names of the IRIs with prefix p_i and is called a partition of $\mathcal{B}^P$.

Consider the example where the IRI `dbr:Copenhagen` is inserted into a PPBF, visualized in Figure F.2a. In this case, the IRI is matched to the prefix `dbr`, and the IRI is hashed using each hash function in the PPBF; each corresponding bit in the bitvector for the `dbr` prefix is thus set to 1.



(a) PPBF $\mathcal{B}_1^P$ **(b)** $\mathcal{B}_1^P \cap \mathcal{B}_2^P$

Fig. F.2: Example of (a) inserting an IRI into a PPBF $\mathcal{B}_1^P$ and (b) intersection between two PPBFs $\mathcal{B}_1^P \cap \mathcal{B}_2^P$ [5].

Like for regular Bloom filters, we say that an IRI i with prefix p *may* be in a PPBF $\mathcal{B}^P$, denoted $i \sqsubseteq \mathcal{B}^P$, if and only if all positions given by $h(i)$ such that $h \in H$ are set to 1 in the bitvector $\theta(p)$. PPBFs are used by Pqnic and ColChain to prune non-overlapping fragments of joining triple patterns from the query execution plan (i.e., the $match(P, I)$ function in Definition F.7). This is done by finding the intersection of the two PPBFs to check whether or not they overlap; if the intersection of the two PPBFs is empty, the corresponding fragments do not produce any join results. The PPBF intersection is defined in [5] as follows.

Definition F.10 (Prefix-Partitioned Bloom Filter Intersection [5])
The intersection of two PPBFs with the same set of hash functions H and bitvectors of the same size, denoted $\mathcal{B}_1^P \cap \mathcal{B}_2^P$, is $\mathcal{B}_1^P \cap \mathcal{B}_2^P = (P_\cap, \hat{B}_\cap, \theta_\cap, H)$, where $P_\cap = \mathcal{B}_1^P.P \cap \mathcal{B}_2^P.P$, $\hat{B}_\cap = \{\mathcal{B}_1^P.\theta(p) \,\&\, \mathcal{B}_2^P.\theta(p) \mid p \in P_\cap\}$, and $\theta_\cap : P_\cap \rightarrow \hat{B}_\cap$.

Consider the example intersection visualized in Figure F.2b. As described above, the intersection of two PPBFs is the bitwise `AND` operation on the bitvectors for the prefixes that $\mathcal{B}_1^P$ and $\mathcal{B}_2^P$ have in common. In this example, $\mathcal{B}_2^P$ does not have a bitvector with the prefix `dbp`, thus this partition is

omitted from the intersection. Similarly, the bitvector partition with the dbo prefix is omitted. Since both PPBFs have bitvectors for the dbr prefix, the resulting PPBF has one partition for the dbr prefix that is a result of the bitwise AND operation between the two corresponding partitions in $\mathcal{B}_1^P$ and $\mathcal{B}_2^P$.

4 The Lothbrok Approach

Differently from Piqnic and ColChain, Lothbrok uses a fragmentation strategy based on characteristic sets. To accommodate efficient query processing over such fragments, as well as to enable locality-awareness and more accurate cardinality estimation, Lothbrok introduces an indexing scheme that maps star patterns to fragments rather than triple patterns. In the remainder of this section, we provide a brief overview of the Lothbrok architecture and how Lothbrok optimizes SPARQL queries over decentralized knowledge graphs, followed by a formal definition of the fragmentation and indexing approach. Query optimization with details on how to exploit locality-awareness and join ordering are explained in Section 5.

4.1 Design and Overview

Lothbrok introduces three contributions, that altogether decrease the communication overhead and in doing so increases query processing performance. First, Lothbrok creates fragments based on characteristic sets such that entire star patterns can be answered by a single fragment. This is beneficial since, as we discussed in Section 1, such star patterns are relatively efficiently processed by the nodes [56] and reduce the communication overhead. The characteristic set of a subject value (entity) is the set of predicates that occur in triples with that subject. As such, Lothbrok creates one fragment per unique characteristic set and each fragment thus contains all the triples with the subjects that match the characteristic set of the fragment. Consider, for instance, the example network in Figure F.1 and query Q shown in Figure F.3a. Table F.3b shows the characteristic sets of each fragment in the network. Using this fragmentation method, each fragment can provide answers to entire star patterns; for instance, $P_3 \in \mathcal{S}(\mathcal{Q})$ can be processed over just f_5, since it is the only fragment containing triples with both predicates present in P_3. The formal definition of the fragmentation approach is presented in Section 4.2.

Second, to accommodate processing entire star patterns over individual fragments, and to encode structural information that can be used for cardinality estimation and locality awareness, Lothbrok introduces a novel indexing scheme, called Semantically Partitioned Blooms Filter (SPBF) Indexes, that builds upon the Prefix-Partitioned Bloom Filter (PPBF) indexes presented in [5]. In particular, SPBFs partition the bitvectors based on the IRI's position

```
select * where {
  ?person dbo: nationality ?country . # tp1 (P1)
  ?person dbo:author ? publication . # tp2 (P1)
  ?country dbo: capital ? capital . # tp3 (P2)
  ?country dbo:currency ?currency . # tp4 (P2)
  ? publication dbo: publisher ? publisher . #tp5 (P3)
  ? publication dbp:language ?language . #tp6 (P3)
}
```

(a) Query Q

Fragment	CS
f_1	{dbo:nationality,dbo:author,dbo:deathDate}
f_2	{dbo:nationality,dbo:author}
f_3	{dbo:capital,dbo:currency,dbo:population}
f_4	{dbo:capital,dbo:currency}
f_5	{dbo:publisher,dbo:language}

(b) CSs of each fragment in the running example

Fig. F.3: (a) Example SPARQL query Q and (b) corresponding characteristic sets in the example network.

in the fragment, i.e., whether it is a subject, predicate, or object. For instance, in the running example, the SPBF for f_5 contains a partition encoding all the subjects with the characteristic set {dbo:publisher,dbo:language}, as well as partitions encoding all the objects in f_5 that occur in a triple with each predicate. The formal definition of SPBF indexes is discussed in Section 4.3.

Third, Lothbrok proposes a query optimization technique that takes advantage of the fragmentation based on characteristic sets and the SPBF indexes to estimate cardinalities and consider data locality while optimizing the query execution plan. First, Lothbrok builds a *compatibility graph* using the SPBF indexes that describes, for a given query, which fragments are compatible with one another for each star join in the query (i.e., which fragments may produce results for the joins). Then, Lothbrok builds a query execution plan using a Dynamic Programming (DP) algorithm that considers the compatibility of fragments in the compatibility graph and the locality of the fragments in the index.

In the remainder of this section, we detail data fragmentation (Section 4.2) and indexing (Section 4.3) in Lothbrok. Section 5 details the query optimization approach used by Lothbrok.

4.2 Data Fragmentation

As discussed in Section 1, star-shaped subqueries can be processed relatively efficiently over a fragment [56], thus they can also help achieving a better balance between reducing the communication overhead and distributing the query processing load [3, 13, 14]. To facilitate processing such star patterns on single nodes, we propose to fragment the uploaded knowledge graphs based on *characteristic sets* [13, 14, 53]. Formally, a characteristic set is defined as follows:

Definition F.11 (Characteristic Set [13, 14, 53])
The characteristic set for a subject s in a given knowledge graph $\mathcal{G}$, $C_{\mathcal{G}}(s)$, is the set of predicates associated with s, i.e., $C_{\mathcal{G}}(s) = \{p \mid (s,p,o) \in \mathcal{G}\}$. The set of characteristic sets of a knowledge graph $\mathcal{G}$ is $C(\mathcal{G}) = \{C_{\mathcal{G}}(s) \mid (s,p,o) \in \mathcal{G}\}$.

In other words, the characteristic set of a subject is the set of predicates (i.e., predicate combination) used to describe the subject, i.e., that occur in the same triples as the subject. For instance, if the triples (`dbr:Denmark`,`dbo:capital`,`dbr:Copenhagen`) and (`dbr:Denmark`,`dbo:currency`,`dbr:Danish_Krone`) are the only ones with subject `dbr:Denmark`, then this subject is described by the characteristic set {`dbo:capital`,`dbo:currency`}.

Characteristic sets were first introduced in [53], used for cardinality estimation and, in extension of that, join ordering. WiseKG [13] and SmartKG [14] used the notion of characteristic sets for fragmentation of knowledge graphs in LDF systems to balance the query load between clients and servers. In this paper, we use characteristic set based fragments as an alternative to the purely predicate-based fragmentation used by for example Piqnic. We define the characteristic set based fragmentation function as follows:

Definition F.12 (Characteristic Set Fragmentation Function)
Let $\mathcal{G}$ be a knowledge graph, then the *characteristic set fragmentation function* of $\mathcal{G}$, $\mathcal{F}_C(\mathcal{G})$, is defined using the notation introduced in Definition F.11, as:

$$\mathcal{F}_C(\mathcal{G}) = \{\{(s,p,o) \mid (s,p,o) \in \mathcal{G} \wedge C_{\mathcal{G}}(s) = C_i\} \mid C_i \in C(\mathcal{G})\} \tag{F.2}$$

That is, the characteristic set fragmentation function creates a fragment for each characteristic set in the knowledge graph. In the characteristic sets shown in Figure F.3b, f_4 thus contains all triples of all subjects that are described by the characteristic set {`dbo:capital`,`dbo:currency`}.

Lothbrok nodes can then use these fragments to process entire star patterns. However, for relatively unstructured knowledge graphs, using fragmentation purely based on characteristic sets can lead to an unwieldy number of fragments. For instance, in our experimental evaluation in Section 7, fragmenting the data from LargeRDFBench [59] using Equation F.2 led to 181,859 distinct fragments, most of which contain very few subjects. Usually, these fragments are created for subjects that are unique due to one or two predicates, while the remaining predicates could fit into larger fragments.

Consider, for instance, in the running example, the situation where the following five characteristic sets are found in the uploaded knowledge graph; for illustration purposes we have extended the notation with the number of subjects covered by each characteristic set:

$$\begin{aligned}
CS_1 &= (\{\texttt{dbo:nationality}, \texttt{dbo:author}, \texttt{dbo:deathDate}\}, 500)\\
CS_2 &= (\{\texttt{dbo:nationality}, \texttt{dbo:author}\}, 500)\\
CS_3 &= (\{\texttt{dbo:publisher}, \texttt{dbo:language}\}, 1000)\\
CS_4 &= (\{\texttt{dbo:nationality}, \texttt{dbo:author}, \texttt{dbo:language}\}, 2)\\
CS_5 &= (\{\texttt{dbo:nationality}\}, 1)
\end{aligned}$$

In this case, a separate fragment is created for CS_4 even though it does not carry very much information because it describes only two subjects. While this is not such a big issue in terms of space, it affects the lookup time when optimizing the join order and estimating the cardinalities, since the query processor has to consider potentially thousands of such small fragments. As such, and similar to [53], we merge infrequent characteristic sets into fragments with a larger number of subjects. After fragmenting datasets using Equation F.2, we apply a strategy with two sequential steps for fragments with infrequent characteristic sets.

First, we merge a fragment f_1 with characteristic set CS_1 into a fragment f_2 with characteristic set CS_2 if $CS_1 \subseteq CS_2$ by adding the triples of f_1 to f_2; if there are multiple candidates for f_2, we select the one with the smallest set of predicates. In the example above, for instance, we merge CS_5 into CS_2 by adding the subject from the fragment with CS_5 to the fragment with CS_2.

Second, we split fragments f with infrequent characteristic sets into two separate fragments, f_1 and f_2, such that f_1 and f_2 can be merged into other fragments with more frequent characteristic sets. In the example above, we thus split the fragment with CS_4 into two smaller fragments f_4' and f_4'' such that f_4' has the characteristic set $\{\texttt{dbo:nationality},\texttt{dbo:author}\}$ and f_4'' has the characteristic set $\{\texttt{dbo:language}\}$; f_4' is then merged into the fragment with CS_2 and f_4'' is merged into the fragment with CS_3. For example, in the example above, we end up with the following fragments:

$$CS_1 = (\{\texttt{dbo:nationality}, \texttt{dbo:author}, \texttt{dbo:deathDate}\}, 500)$$
$$CS_2 = (\{\texttt{dbo:nationality}, \texttt{dbo:author}\}, 503)$$
$$CS_3 = (\{\texttt{dbo:publisher}, \texttt{dbo:language}\}, 1002)$$

4.3 Semantically Partitioned Bloom Filter Indexes

The indexing schema presented in [5] (Definition F.6) represents the set of subject and object values as prefix-partitioned bitvectors based on Bloom filters [15] called Prefix-Partitioned Bloom Filters (PPBFs). However, PPBFs encode the entire set of subjects and objects in a fragment as a single set and ignore the position (subject or object) of the IRIs in the triples; as such, in a situation where two fragments, for instance, use the same IRIs in the object position, the intersection of the two PPBFs is non-empty. Then, if the corresponding triple patterns in the query are joined with a subject-object join, the fragments are not pruned since the PPBFs overlap; however, since we are looking for a subject-object join rather than an object-object join, these fragments could have been pruned without affecting the query completeness. Furthermore, PPBF indexes do not include the predicate values in the index slices, rather they associate the predicate value with the index slice itself

(Definition F.9), thus maintaining information about the links between the subjects, predicates, and objects. This is possible since the implementations of Piqnic and ColChain use the predicate-based fragmentation function; however, Lothbrok allows for fragments with several distinct predicates.

Hence, to efficiently estimate whether or not fragments join for a particular query and to maintain the connection between the subjets, predicates, and objects for fragments with multiple predicates, we propose an indexing schema called *Semantically Partitioned Bloom Filters* (SPBFs), which builds upon PPBF as baseline. As the triples contained in fragments defined based on characteristic sets (Section 4.2) share the same subjects, SPBFs encode the subject values in a single prefix-partitioned bitvector, while there is one prefix-partitioned bitvector for each predicate in the fragment that encodes the objects occurring in triples with that predicate. For instance, in the running example, each subject within f_2 occurs in triples with both `dbo:nationality` and `dbo:author` as predicates. The SPBF for f_2 contains one partition describing the subject values, one partition describing the object values connected with the `dbo:nationality` predicate, and one partition describing the object values connected with the `dbo:author` predicate. Formally, an SPBF is defined as follows:

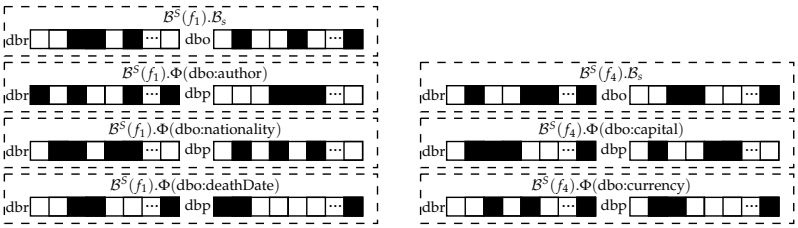
Definition F.13 (Semantically Partitioned Bloom Filter)
An SPBF $\mathcal{B}^S$ is a 5-tuple $\mathcal{B}^S = (P, \mathcal{B}_s, B_o, \Phi, H)$ where:

- P is a set of distinct predicate values
- $\mathcal{B}_s$ is the prefix-partitioned bitvector that summarizes the subjects
- B_o is the set of prefix-partitioned bitvectors that summarize the objects
- $\forall \mathcal{B}_i \in \{\mathcal{B}_s\} \cup B_o$, $\mathcal{B}_i = (P_i, \hat{B}_i, \theta_i)$ where:
  - P_i is a set of prefixes
  - $\hat{B}_i$ is a set of bitvectors such that $\forall \hat{b}_1, \hat{b}_2 \in \hat{B}_i : |\hat{b}_1| = |\hat{b}_2|$
  - $\theta_i : P_i \rightarrow \hat{B}_i$ is a prefix-mapping function
- $\Phi : P \rightarrow B_o$ is a predicate-mapping function s.t. $\forall p \in P : \Phi(p) \in B_o$
- H is a set of hash functions

Similarly to prefix-partitioned bitvectors, we say that an IRI i at position $\rho \in \{s, p, o\}$ *may* be in an SPBF $\mathcal{B}^S$, denoted $i \sqsubset^{\rho} \mathcal{B}^S$, if and only if $i \sqsubset \mathcal{B}^S.\mathcal{B}_s$ if $\rho = s$, $\exists p \in \mathcal{B}^S.P : i \sqsubset \mathcal{B}^S.\Phi(p)$ if $\rho = o$, or $i \in \mathcal{B}^S.P$ if $\rho = p$. Furthermore, $\mathcal{B}_p(\mathcal{B}^S)$ denotes a function that computes and returns the prefix-partitioned bitvector that contains all predicates in $\mathcal{B}^S.P$. Given a fragment f, $\mathcal{B}^S(f)$ describes the SPBF for f.

Consider again the running example from Figure F.1. Figure 22 shows the SPBFs of fragments f_1 (Figure F.4a) and f_4 (Figure F.4b). The SPBF for f_1 contains a prefix-partitioned bitvector that encodes all the subject values in f_1, $\mathcal{B}^S(f_1).\mathcal{B}_s$, as well as a prefix-partitioned bitvector for each predicate that encodes the object values that are connected with the predicates, i.e., the

partition $\mathcal{B}^S(f_1).\Phi(\texttt{dbo:author})$ that describes the objects that are connected with the `dbo:author` predicate, and so on. Similar for the SPBF for f_4, $\mathcal{B}^S(f_4)$.



(a) SPBF $\mathcal{B}^S(f_1)$ **(b)** SPBF $\mathcal{B}^S(f_4)$

Fig. F.4: SPBFs of f_1, $\mathcal{B}^S(f_1)$ (a) and f_4, $\mathcal{B}^S(f_4)$ (b) in the running example.

A distributed index as defined in Definition F.6 and [5, 6] associates triple patterns in the query with fragments that contain relevant data to the triple patterns. However, since LOTHBROK partitions data based on characteristic sets, we adapt the definition of a distributed index to the fragmentation based on characteristic sets and SPBF indexes. Let $relevantFragment(P, f)$ be a function that returns `true` if $\forall t = (s, p, o) \in P$, $s \in V$ or $s \sqsubset^s \mathcal{B}^S(f)$, $p \in V$ or $p \sqsubset^p \mathcal{B}^S(f)$, and $o \in V$ or $o \sqsubset \mathcal{B}^S(f).\Phi(p)$, or `false` otherwise. We define an SPBF index as follows:

Definition F.14 (Semantically Partitioned Bloom Filter Index [5, 6])
Let n be a node and $\mathcal{N}$ be the set of nodes within n's local view of the network, $\mathcal{P}$ be the set of all possible star patterns, and $\mathcal{F}$ be the set of fragments stored by at least one node in $\mathcal{N}$. The *SPBF index* on n is a tuple $I_n^S = (\upsilon, \eta)$ with $\upsilon : \mathcal{P} \mapsto 2^{\mathcal{F}}$ and $\eta : \mathcal{F} \mapsto 2^{\mathcal{N}}$. $\upsilon(P)$ returns the set of fragments F such that $\forall f \in F$, $relevantFragment(P, f) = \texttt{true}$. $\eta(f)$ returns the set of nodes N such that $f \in n_i.G$, $\forall n_i \in N$ and $n_i \in \mathcal{N}$.

In other words, an SPBF index maps a star pattern to the fragments that may contain all the constants within the star pattern, and the fragments to the nodes that store them. Furthermore, since LOTHBROK, like PIQNIC and COLCHAIN, builds partial indexes, i.e., slices, for each fragment that are combined to form the node's distributed index, we define an SPBF index slice as follows:

Definition F.15 (SPBF Slice)
Let f be a fragment. The *SPBF slice* describing f is a tuple $s_f^S = (\upsilon', \eta')$ where $\upsilon'(P)$ returns $\{f\}$ if and only if $relevantFragment(P, f) = \texttt{true}$, and $\eta'(f)$ returns the set of all nodes that contain f in its local datastore.

The function $s^S(f)$ finds the SPBF slice describing f. The SPBF slice describing a fragment is the SPBF obtained from the respective fragment. For instance, in the running example, the SPBF slice of f_1 corresponds to the

SPBF obtained from f_1, i.e., the one in Figure F.4a. In Section 5, we detail how SPBF indexes are used to optimize queries using cardinality estimations and the locality of the data.

5 Query Optimization

To optimize queries over the network, LOTHBROK first determines which fragments are compatible i.e., produce join results for the given query. To do this, LOTHBROK builds a graph that includes the fragments that are compatible for star patterns in the given query, called a *compatibility graph*. In other words, the nodes in a compatibility graph are fragments, and the edges connect the compatible ones.

Compatibility graphs encapsulate two things. First, the fragments within a compatibility graph are the fragments that contribute to the overall query result, i.e., fragments that do not contribute to the result are pruned. Second, different branches of a compatibility graph for the same subqueries can be processed in parallel. Take, for instance, again query Q in Figure F.3a. In this case, assuming the join order $P_2 \bowtie P_1 \bowtie P_3$ (details on join order optimization in Section 5.3) and the compatibility of the fragments given in Figure F.5g (details on compatibility graphs in Section 5.1), then subquery $P_2 \bowtie P_1$ could be processed concurrently over $\{f_1, f_4\}$ and $\{f_2, f_3\}$ since f_1 only depends on the intermediate results from f_4 and f_2 only depends on the intermediate results from f_3. Hence, it could be beneficial to process $P_1 \cup P_2$ by delegating the subquery to nodes n_2 and n_3 concurrently such that n_2 processes $[[P_2]]_{f_4} \bowtie [[P_1]]_{f_1}$ locally and n_3 processes $[[P_2]]_{f_3} \bowtie [[P_1]]_{f_2}$ locally, and using the combined (by union) results as intermediate bindings when processing $[[P_3]]_{f_5}$ on node n_1.

To this end, LOTHBROK applies Dynamic Programming (DP) similar to [50, 53] to build a query execution plan specifying join delegations and parallel processing of subqueries. To further decrease the network overhead, we adapt the cost function in the DP algorithm to consider data *locality* and cardinality estimations available using the SPBF indexes. In other words, the cost function estimates, given a query execution plan, how many intermediate results processing the join on a particular node incurs, and selects the execution plan that incurs the least data transfer overhead.

In summary, given a BGP P, LOTHBROK optimizes P by applying the following steps:

1. Select the relevant fragments for each star pattern in P using the SPBF index.
2. Build the compatibility graph G^C for P (Section 5.1) by checking the overlap of the corresponding bitvector partitions in the SPBF index.

3. Build a query execution plan using Dynamic Programming (DP) on P and G^C in consideration of cardinality estimations (Section 5.2) and data locality (Section 5.3).

The output of the above steps is a query execution plan. In the remainder of this section, we go into details with source selection using compatibility graphs, cardinality estimation, and the query optimization strategy using Dynamic Programming. In Section 6, we describe how a query execution plan is processed.

5.1 Fragment and Source Selection

As mentioned above, query optimization in Lothbrok exploits fragment *compatibility*. To achieve this, nodes build a *compatibility graph* describing which fragments are compatible for a given query. Two fragments are said to be compatible for a given query if the intersection of the corresponding SPBF partitions is non-empty. A compatibility graph is thus an undirected graph where nodes are the relevant fragments for the star patterns in the query (determined using the SPBF index) and edges describe the compatible ones.

Recall the function $\mathcal{B}^S(f)$ that returns the SPBF for a fragment f, and let $vars(P)$ be a function that returns all the variables in a star pattern P. Furthermore, given an SPBF $\mathcal{B}^S$, a star pattern P, and a variable v, let $\mathcal{B}(\mathcal{B}^S, P, v)$ denote a function that returns (assuming v can only occur once in P) $\mathcal{B}^S.\mathcal{B}_s$ if v is the subject in P, $\mathcal{B}^S.\Phi(p)$ if v is the object with predicate p, i.e., $(s, p, v) \in P$, or $\mathcal{B}_p(\mathcal{B}^S)$ if v is a predicate in P. Then, a compatibility graph of a BGP P and SPBF index I^S is formally defined as follows.

Definition F.16 (Compatibility Graph)
Given an SPBF index I^S and a BGP P, the *compatibility graph* G^C of P over I^S is a tuple $G^C(P, I^S) = (F, C)$ such that $\forall P_1, P_2 \in \mathcal{S}(P)$ where $vars(P_1) \cap vars(P_2) \neq \emptyset$ and $\forall v \in vars(P_1) \cap vars(P_2)$, it is the case that $\forall f_1 \in I^S.v(P_1), f_2 \in I^S.v(P_2)$ where $\mathcal{B}(\mathcal{B}^S(f_1), P_1, v) \cap \mathcal{B}(\mathcal{B}^S(f_2), P_2, v) \neq \emptyset$, $(f_1, f_2) \in C$ and $f_1, f_2 \in F$. Furthermore, $\forall P' \subseteq P$ where $vars(P') \cap vars(P - P') = \emptyset$ (i.e., for Cartesian products), it is the case that $\forall f_1 \in F$ such that $f_1 \in I^S.v(P_1)$ for some $P_1 \in \mathcal{S}(P')$ and $\forall f_2 \in F$ such that $f_2 \in I^S.v(P_2)$ for some $P_2 \in \mathcal{S}(P - P')$, $(f_1, f_2) \in C$.

For instance, in the running example, let $\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s \neq \emptyset$, i.e., f_1 and f_4 produce join results, and $\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_3).\mathcal{B}_s = \emptyset$, i.e., f_1 and f_3 do not overlap. Then, the compatibility graph for query Q in Figure F.3a contains an edge between f_1 and f_4, but no edge between f_1 and f_3. We denote the empty compatibility graph (i.e., where F and C are empty sets) as $G^C_\emptyset$. Algorithm 12 defines the $G^C(P, I^S)$ function in lines 1-16 that computes a compatibility graph given a BGP P and SPBF index I^S.

Algorithm 12 Compute the Compatibility Graph of a BGP over an SPBF index

Input: A BGP $P = P_1 \cup \cdots \cup P_n$; an SPBF index $I^S = (v, \eta)$
Output: A compatibility graph G^C

1: **function** $G^C(P, I^S)$
2: $P' \leftarrow P_k$ where $P_k \in \mathcal{S}(P)$ and $card_B(P_k) \leq card_B(P_j) \forall P_j \in \mathcal{S}(P)$;
3: $P_\epsilon \leftarrow P'$;
4: $F, C \leftarrow \emptyset$;
5: **for all** $f \in I^S.v(P')$ **do**
6: $G^C_\epsilon \leftarrow$ `buildBranch`$(P - P', I^S, f, P', P_\epsilon)$;
7: $F \leftarrow F \cup G^C_\epsilon.F$;
8: $C \leftarrow C \cup G^C_\epsilon.C$;
9: **if** $P - P_\epsilon \neq \emptyset$ **then**
10: $G^C_\epsilon \leftarrow \mathtt{G}^C(P - P_\epsilon, I^S)$;
11: **if** $G^C_\epsilon = G^C_\emptyset$ **then return** $G^C_\emptyset$
12: **for all** $f_1 \in F, f_2 \in G^C_\epsilon.F$ **do**
13: $C \leftarrow C \cup \{(f_1, f_2)\}$;
14: $F \leftarrow F \cup G^C_\epsilon.F$;
15: $C \leftarrow C \cup G^C_\epsilon.C$;
16: **return** (F, C);
17: **function** *buildBranch*$(P, I^S, f, P', P_\epsilon)$
18: **if** $P = \emptyset$ **or** $\forall P'' \in \mathcal{S}(P) : vars(P') \cap vars(P'') = \emptyset$ **then**
19: **return** $(\{f\}, \emptyset)$;
20: $F, C \leftarrow \emptyset$;
21: **for all** $P'' \in \mathcal{S}(P)$ s.t. $vars(P') \cap vars(P'') \neq \emptyset$ **do**
22: $P'_\epsilon \leftarrow P_\epsilon \cup P''$;
23: $V \leftarrow vars(P') \cap vars(P'')$;
24: **for all** $f' \in I^S.v(P'')$ s.t. $\forall v \in V : \mathcal{B}(\mathcal{B}^S(f), P', v) \cap \mathcal{B}(\mathcal{B}^S(f'), P'', v) \neq \emptyset$ **do**
25: $G^C_\epsilon \leftarrow$ `buildBranch`$(P - P'', I^S, f', P'', P'_\epsilon)$;
26: **if** $G^C_\epsilon \neq G^C_\emptyset$ **then**
27: $F \leftarrow F \cup G^C_\epsilon.F \cup \{f\}$;
28: $C \leftarrow C \cup G^C_\epsilon.C \cup \{(f, f')\}$;
29: $P_\epsilon \leftarrow P_\epsilon \cup P'_\epsilon$;
30: **return** (F, C);

Figure F.5 shows how Algorithm 12 builds the compatibility graph for query Q in Figure F.3a. In the following, we go through each intermediate step of the algorithm, describing the intermediate compatibility graphs built in the process. First, the G^C function selects the star pattern in $\mathcal{S}(P)$ with the lowest estimated cardinality in line 2 (cardinality estimation is detailed

in Section 5.2). Assume in the running example, that P_2 is the star pattern with the lowest estimated cardinality (Section 5.2), and that it is therefore selected in line 2 as the first star pattern. Furthermore, assume that f_1 is only compatible with f_4 and f_2 is compatible with f_3.

Then, the relevant fragments for the selected star pattern are found using the $I^S.v$ function from the SPBF index (Definition F.14) and iterated over in the for loop in lines 5-8; for each of these fragments, the function calls the `buildBranch`$(P, I^S, f, P', P_\epsilon)$ function in lines 17-30 that builds the (sub)graph starting from the current fragment. In the example, the loop in lines 5-8 iterates over $\{f_3, f_4\}$, since these are the fragments relevant for P_2.

The `buildBranch`$(P, I^S, f, P', P_\epsilon)$ function defines a recursive function that builds a sub-graph starting from a specific fragment and star pattern. In the first iteration in the running example (i.e., for f_3), `buildBranch` is called with $P = P_1 \cup P_3$, $f = f_3$, and $P' = P_2$ as parameters. First, if P does not contain any star patterns that join with P', i.e., if P' is the outer-most star pattern in the join tree or for a Cartesian product, the function returns the compatibility graph just containing f without any edges (lines 18-19). In the example, since P_1 joins with P_2, the algorithm does not enter the if statement in line 19.

Instead, the for loop in lines 21-29 iterates through the star patterns $P'' \in P$ that join with P', i.e., star patterns that have at least one variable in common. For each fragment f' relevant for P'' (again found using the SPBF index), the function checks the compatibility of f and f' for each join variable v in line 24, i.e., whether or not f and f' may produce join results for each join variable, by intersecting the corresponding partitioned bitvectors in $\mathcal{B}^S(f)$ and $\mathcal{B}^S(f')$. If the fragments may produce join results, a recursive call is made in line 25 with the $P = P - P''$, $f = f'$, and $P' = P''$ as parameters. In the example, the for loop in line 21 has only one iteration for $P'' = P_1$, i.e., the only star pattern in $\mathcal{S}(P)$ that joins with P_2. Hence, the for loop in line 24 checks the compatibility of each fragment relevant for P_1 (f_1 and f_2) with f_3 (since $f = f_3$ in this call to the function). Since f_2 is compatible with f_3 (cf. the join cardinalities in Table F.1), a recursive call is made in line 25 with $P = P_3$, $f = f_2$, and $P' = P_1$.

Since P_3 joins with P_1, the for loop in line 24 checks the compatibility of f_5 and f_2 and makes another recursive call to the function in line 25 with $P = \emptyset$, $f = f_5$, and $P' = P_3$. In this iteration of the function, P is empty, thus the graph $(\{f_5\}, \emptyset)$ is returned in line 19. This graph is visualized in Figure F.5a and contains only f_5 with no edges. Since this compatibility graph is non-empty, it is added to the output graph in lines 26-28 together with f_2 (since $f = f_2$ in this iteration of `buildBranch`) and the edge between f_5 and f_2. This graph is visualized in Figure F.5b and returned by the current iteration of the `buildBranch` function. Upon receiving the graph in Figure F.5b, the function adds f_3 (since $f = f_3$ in the current iteration) and an edge between f_2 and f_3 in lines 26-28, resulting in the compatibility graph shown on Figure F.5c that

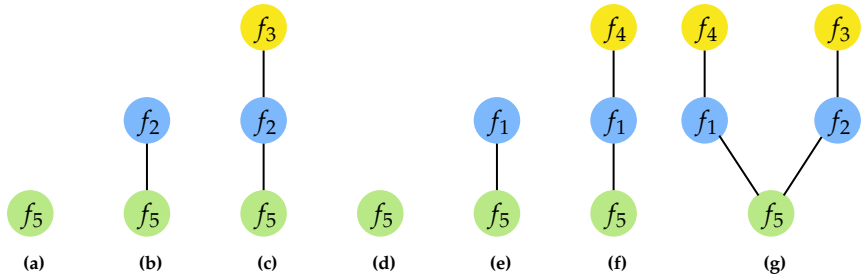


Fig. F.5: Recursively building the compatibility graph for the query in Figure F.3a by applying Algorithm 12 resulting in $G^C(Q, I^S_{n_1})$. Yellow nodes denote the fragments relevant for P_2, blue nodes the fragments relevant for P_1, and the green nodes the fragments relevant for P_3.

is returned in line 30.

In the next iteration of the for loop in line 5, the `buildBranch` is called with $P = P_1 \cup P_3$, $f = f_4$, and $P' = P_2$. Following the same procedure as described above for f_3, we first build the subgraph containing only f_5 shown in Figure F.5d. Then, f_1 is added to the graph along with an edge between f_1 and f_5 (since they produce join results), resulting in the subgraph shown in Figure F.5e. Next, f_4 is added along with an edge between f_4 and f_1, resulting in the compatibility graph for f_4 shown in Figure F.5f. After merging this in lines 7-8 with the compatibility graph in Figure F.5c, the resulting compatibility graph can be seen in Figure F.5g.

The if statement in lines 9-15 ensures that subqueries with star patterns that do not join (i.e., in the case of Cartesian products) are included in the compatibility graph. This is done by keeping track of the considered star patterns in P using the accumulator P_c defined in line 3 and updated in line 29. The example query contains no Cartesian products and so the compatibility graph on Figure F.5g is returned by the algorithm.

The output of Algorithm 12 in the example is the compatibility graph shown in Figure F.5g, specifying that f_1 is compatible with $\{f_4, f_5\}$ and f_2 is compatible with $\{f_3, f_5\}$.

5.2 Cardinality Estimation

In Section 4.2 we have described how Lothbrok fragments knowledge graphs based on characteristic sets. Furthermore, in Section 4.3 we described how SPBF indexes connect the objects in a fragment to the predicates they occur in triples with. Since the SPBF of a fragment includes partitioned bitvectors describing the subjects and objects (Definition F.14), we can estimate the number of values within these partitioned bitvectors and use those estimations to obtain cardinality estimations in a similar way as [50, 53]. To achieve this, we first define the estimated number of values in a partitioned bitvector.

Given a partitioned bitvector $\mathcal{B}$ and $\hat{b} \in \mathcal{B}.\hat{B}$, let $t(\hat{b})$ be a function that returns the number of bits in $\hat{b}$ that are set. Then, the estimated cardinality of a partitioned bitvector $\mathcal{B}$, denoted $card^P(\mathcal{B})$, is the sum of the estimated cardinality for all bitvector partitions in $\mathcal{B}.\hat{B}$ [5, 54] and is formally defined as follows:

$$card^P(\mathcal{B}) = \sum_{\hat{b} \in \mathcal{B}.\hat{B}} \frac{ln(1 - t(\hat{b})/|\hat{b}|)}{|\mathcal{B}.H| \cdot ln(1 - 1/|\hat{b}|)} \tag{F.3}$$

Consider, for instance, again the running example introduced in Section 3.1 and the SPBF for f_4, $\mathcal{B}^S(f_4)$, in Figure F.4b. Assume that $|\mathcal{B}^S(f_4).H| = 5$ and that $|\hat{b}| = 20000$ for all $\hat{b} \in \mathcal{B}^S(f_4).\hat{B}$. Since the partitioned bitvector for the predicate `dbo:capital` in f_4 (Figure F.4b) has two partitions, `dbr` and `dbp`, obtaining the estimated cardinality for $\mathcal{B}^S(f_4).\Phi(\texttt{dbo:capital})$ is the sum of estimating the cardinality of both prefix partitions. Let the number of set bits in the bitvector for the `dbr` prefix be 736 and the number of set bits in the bitvector for the `dbp` prefix be 249. Then, the estimated cardinality using Equation F.3 is:

$$card^P(\mathcal{B}^S(f_4).\Phi(\texttt{dbo:capital})) = \frac{ln(1 - 736/20000)}{5 \cdot ln(1 - 1/20000)} + \frac{ln(1 - 249/20000)}{5 \cdot ln(1 - 1/20000)} \approx \frac{-0.0375}{-0.00025} + \frac{-0.0125}{-0.00025} \approx 150 + 50 \approx 200$$

Table F.1: Estimated cardinalities for the SPBFs $\mathcal{B}^S(f_1)$, $\mathcal{B}^S(f_2)$, $\mathcal{B}^S(f_3)$, and $\mathcal{B}^S(f_4)$ for the running example in Figure F.1

Partitioned Bitvector	$card^P$	Partitioned Bitvector	$card^P$
$\mathcal{B}^S(f_1).\mathcal{B}_s$	1000	$\mathcal{B}^S(f_3).\mathcal{B}_s$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:author})$	5000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:capital})$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality})$	1000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:currency})$	150
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:deathDate})$	1000	$\mathcal{B}^S(f_3).\Phi(\texttt{dbo:population})$	100
$\mathcal{B}^S(f_2).\mathcal{B}_s$	2000	$\mathcal{B}^S(f_4).\mathcal{B}_s$	200
$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:author})$	3000	$\mathcal{B}^S(f_4).\Phi(\texttt{dbo:capital})$	200
$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality})$	2000	$\mathcal{B}^S(f_4).\Phi(\texttt{dbo:currency})$	500
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_3).\mathcal{B}_s$	0	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_3).\mathcal{B}_s$	100
$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s$	50	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s$	0
$\mathcal{B}^S(f_5).\mathcal{B}_s$	8000	$\mathcal{B}^S(f_1).\Phi(\texttt{dbo:author}) \cap \mathcal{B}^S(f_5).\mathcal{B}_s$	500
$\mathcal{B}^S(f_5).\Phi(\texttt{dbo:publisher})$	8000	$\mathcal{B}^S(f_2).\Phi(\texttt{dbo:author}) \cap \mathcal{B}^S(f_5).\mathcal{B}_s$	1000
$\mathcal{B}^S(f_5).\Phi(\texttt{dbo:language})$	9000		

Table F.1 shows the estimated cardinalities of each partitioned bitvector in the running example.

To estimate the cardinality of star-shaped subqueries, we utilize the fact that the subjects are described by a single partitioned bitvector. For a star-shaped subquery asking for the set of unique subject values described by a

given set of predicates (i.e., queries with the DISTINCT keyword), the cardinality can be estimated as the sum of the number of subjects in each fragment that includes all the predicates in the query. For instance, the cardinality of P_1 in the query in Figure F.3a is the number of distinct subject values in f_1 and f_2.

Given a star pattern P and a fragment f, the cardinality of P over f, assuming that f is a relevant fragment for P, is the number of values in the partitioned bitvector on the subject position in $\mathcal{B}^S(f)$, and is formally defined as:

$$card_D(P, f) = card^P(\mathcal{B}^S(f).\mathcal{B}_s) \tag{F.4}$$

For queries not including the DISTINCT keyword, we need to account for duplicates by considering, on average, the number of triples for each non-variable predicate value in P that each subject value is associated with. Given a star pattern P and fragment f, let $preds(P)$ denote the non-variable predicate values in P (in the case of a variable on the predicate position in P, we consider the average number of predicate occurences in the characteristic set). The cardinality of P is thus estimated as follows [50, 53]:

$$card_S(P, f) = card_D(P, f) \cdot \prod_{p_i \in preds(P)} \frac{card^P(\mathcal{B}^S(f).\Phi(p_i))}{card^P(\mathcal{B}^S(f).\mathcal{B}_s)} \tag{F.5}$$

Henceforth, we will refer to the more generalized function *card* rather than $card_D$ and $card_S$ to be equivalent to $card_D$ for queries with the DISTINCT modifier and $card_S$ for queries without. Using Equations F.4 or F.5, the cardinality of a star pattern P over a node n's SPBF index is, for all queries (both with and without the DISTINCT keyword), the aggregated cardinality over each relevant fragment to P, and is formally defined as follows:

$$card_n(P) = \sum_{f \in I_n^S.\eta(P)} card(P, f) \tag{F.6}$$

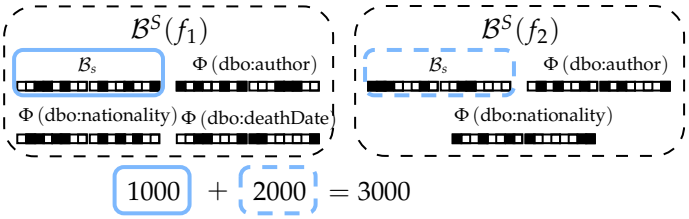


Fig. F.6: Estimating the cardinality of P_1 with the DISTINCT modifier as the number of subjects in f_1 and f_2 found using Equation F.4.

Consider, for instance, in the running example, the star-shaped BGP P_1 in Figure F.3a and the estimated cardinalities of the partitioned bitvectors for each fragment in Table F.1. Assume in this case that the DISTINCT keyword is given in the query. Then, $card_{n_1}(P_1)$ is computed as the aggregated estimation of subject values in f_1 and f_2, $card_{n_1}(P_1) = 1000 + 2000 = 3000$. This is visualized in Figure F.6.

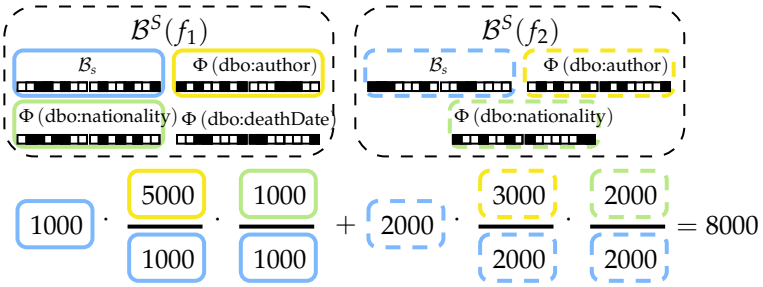


Fig. F.7: Estimating the cardinality of P_1 without the DISTINCT modifier. Outlines show which bitvector each value is computed from.

If, instead, the DISTINCT keyword was not included in the query, the cardinality $card_{n_1}(P_1)$ is, for each relevant fragment (f_1 and f_2), the number of subject values within the fragment multiplied with the average number of triples with each predicate $p_i \in preds(P_1)$ that each subject value is associated with, $card_{n_1}(P_1) = 1000 \cdot (5000/1000) \cdot (1000/1000) + 2000 \cdot (3000/2000) \cdot (2000/2000) = 5000 + 3000 = 8000$. Figure F.7 visualizes the above computations and shows which bitvector each value is computed from.

Until now, the cardinality estimations presented in this section are useful for estimating the cardinality of individual star patterns in a query [50, 53]. However, to estimate the cardinality of arbitrary BGPs, [27] introduced characteristic pairs that describe the connections between IRIs described by different characteristic sets. In our case, however, we rely on the SPBFs of the relevant fragments to compute characteristic pairs without storing additional information; by intersecting the partitioned bitvectors on the positions corresponding to the join variable, we can estimate the selectivity of a given join and use that to estimate the cardinality of the join.

Formally, for queries including the DISTINCT keyword, given two star patterns P_k and P_l that join on a variable v such that $(s, p, v) \in P_k$ and v is the subject of all triple patterns in P_l, and two fragments f_k and f_l that are relevant for P_k and P_l respectively, the cardinality of the join is estimated as the number of IRIs on the subject position in f_k multiplied by the selectivity of the join, i.e., the chance that each subject in the right side corresponds to a value in the join. This is defined as follows:

$$card_D(P_k, P_l, p, f_k, f_l) = card^P(\mathcal{B}^S(f_k).\mathcal{B}_s) \cdot \left(\frac{card^P(\mathcal{B}^S(f_k).\Phi(p) \cap \mathcal{B}^S(f_l).\mathcal{B}_s)}{card^P(\mathcal{B}^S(f_k).\Phi(p))} \right) \quad \text{(F.7)}$$

For queries that do not include the `DISTINCT` keyword, we again consider the average predicate occurrences for each triple pattern in both P_k and P_l, similar to Equation F.5 [27, 50]:

$$card_S(P_k, P_l, p, f_k, f_l) = card_D(P_k, P_l, p, f_k, f_l) \cdot \prod_{p_k \in P_k - \{p\}} \left(\frac{card^P(\mathcal{B}^S(f_k).\Phi(p_k))}{card^P(\mathcal{B}^S(f_k).\mathcal{B}_s)} \right) \cdot \prod_{p_l \in P_l} \left(\frac{card^P(\mathcal{B}^S(f_l).\Phi(p_l))}{card^P(\mathcal{B}^S(f_l).\mathcal{B}_s)} \right) \quad \text{(F.8)}$$

Processing a join between two star patterns over a node n's SPBF is the aggregated cardinality over each pair of relevant fragments to the two star patterns, and is for all queries formally defined as follows:

$$card_n(P_k, P_l, p) = \sum_{f_k \in I_n^S.\eta(P_k) \wedge f_l \in I_n^S.\eta(P_l)} card(P_k, P_l, p, f_k, f_l) \quad \text{(F.9)}$$

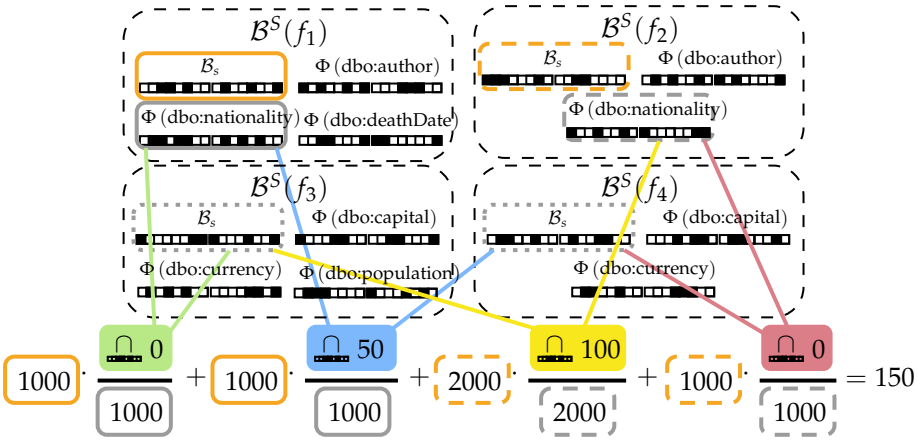
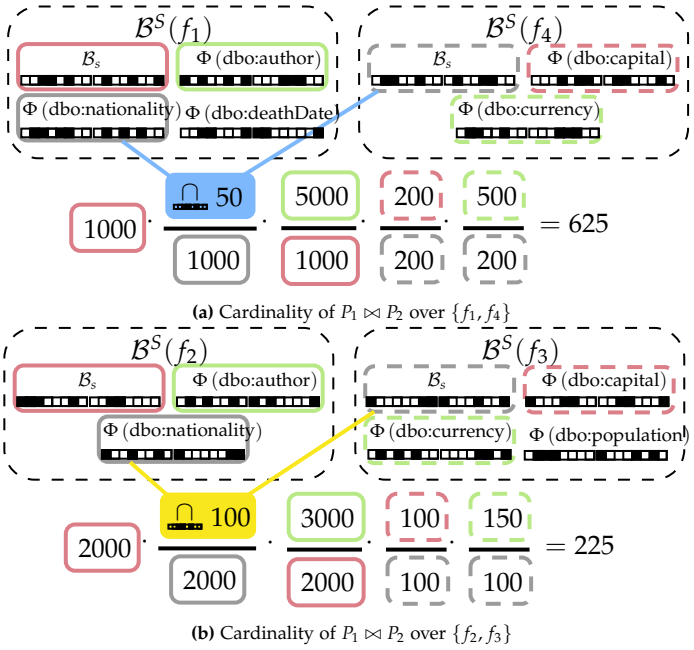


Fig. F.8: Estimating the cardinality of $P_1 \bowtie P_2$ with the `DISTINCT` modifier.

For instance, consider the join between P_1 and P_2 in Figure F.3a where the `DISTINCT` keyword is given in the query. Here, the cardinality $card_{n_1}(P_1, P_2, \texttt{dbo:nationality})$ is the aggregated cardinality of the partitioned bitvectors obtained by intersecting the partitioned bitvector on

the object position for the `dbo:nationality` for each $f_k \in \{f_1, f_2\}$ with the partitioned bitvector on the subject position for each $f_l \in \{f_3, f_4\}$. That is, given the cardinalities of the intersections shown in Table F.1, $card_{n_1}(P_1, P_2, \texttt{dbo:nationality}) = 1000 \cdot (0/1000) + 1000 \cdot (50/1000) + 2000 \cdot (100/2000) + 2000 \cdot (0/1000) = 150$. We have visualized this computation in Figure F.8.



(a) Cardinality of $P_1 \bowtie P_2$ over $\{f_1, f_4\}$

(b) Cardinality of $P_1 \bowtie P_2$ over $\{f_2, f_3\}$

Fig. F.9: Estimating the cardinality of $P_1 \bowtie P_2$ without the `DISTINCT` modifier over (a) $\{f_1, f_4\}$ and (b) $\{f_2, f_3\}$. The output of Equation F.9 is thus the sum of the two formulas ($625 + 225 = 850$).

In the case where the `DISTINCT` keyword is not included in the query, the join between P_1 and P_2 in Figure F.3a given the partitioned bitvector cardinalities in Table F.1 yields the following equation: $card_{n_1}(P_1, P_2, \texttt{dbo:nationality}) = 1000 \cdot (50/1000) \cdot (5000/1000) \cdot (200/200) \cdot (500/200) + 2000 \cdot (100/2000) \cdot (3000/2000) \cdot (100/100) \cdot (150/100) = 625 + 225 = 850$. Figure F.9 visualizes the above computation; Figure F.9a shows the computation over $\{f_1, f_4\}$ and Figure F.9b over $\{f_2, f_3\}$. The outlines show which partitioned bitvector is used to compute each number, e.g., the value 50 is found by computing the cardinality of the bitvector intersection $\mathcal{B}^S(f_1).\Phi(\texttt{dbo:nationality}) \cap \mathcal{B}^S(f_4).\mathcal{B}_s$. The result is the sum of the formulas.

5.3 Optimizing Query Execution Plans

Based on the compatibility graph (Section 5.1), the locality of fragments, and cardinality estimations (Section 5.2), Lothbrok builds a *query execution plan* that specifies which subqueries can be processed in parallel and which joins are delegated to which nodes as well as the join order. A query execution plan is defined as follows:

Definition F.17 (Query Execution Plan)
A *query execution plan* Π consists of the execution plan and the node that processes the plan, called a *delegation*. A query execution plan can be one of four types:

- *Join* $\Pi = \Pi_1 \bowtie^n \Pi_2$ where Π_1 and Π_2 are two (sub)plans and n is the node the join is delegated to.
- *Cartesian product* $\Pi = \Pi_1 \times^n \Pi_2$ where Π_1 and Π_2 are two (sub)plans and n is the node the Cartesian product is delegated to.
- *Union* $\Pi = \Pi_1 \cup^n \Pi_2$ where Π_1 and Π_2 are two (sub)plans and n is the node the union is delegated to.
- *Selection* $\Pi = [[P]]^n_f$ where P is a star pattern, f is the fragment that P is processed over, and n is the node the selection is delegated to.

Since unions are not explicitly executed by any node, instead the partial results of each subplan in the union are transferred to the nodes that use those intermediate results, we simply omit the specification of delegations for unions from the description below. Furthermore, we assume that query execution plans are always left-deep, i.e., the right side of a join can only consist of a selection or a union of selections. For instance, the execution plan for query Q, $\Pi = ((([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$ (Figure F.12g) specifies that the join $[[P_2]]_{f_4} \bowtie [[P_1]]_{f_1}$ is delegated to n_2 and processed in parallel with $[[P_2]]_{f_3} \bowtie [[P_1]]_{f_2}$ on n_3 (specified by the union), the result of which is transferred to n_1 and joined with $[[P_3]]_{f_5}$.

Since our cost function includes the estimated cardinality of a particular subplan, we first extend the framework for cardinality estimation described in Section 5.2 to enable cardinality estimation of an entire query execution plan. This is straightforward for Cartesian products, unions, and selections; for Cartesian products it is the multiplication of the cardinality of the operands, for unions it is the sum of the cardinality of the operands, and for selections it is the cardinality of the star pattern over a specific fragment defined in Equations F.4 and F.5. Given the reasoning above, we define the cardinality of a query execution plan Π, $card(\Pi)$, covering all types of Π, as follows:

$$card(\Pi) = \begin{cases} card(\Pi_1) \cdot card(\Pi_2), & \text{if } \Pi = \Pi_1 \times^n \Pi_2 \\ card(\Pi_1) + card(\Pi_2), & \text{if } \Pi = \Pi_1 \cup \Pi_2 \\ card(P, f), & \text{if } \Pi = [[P]]^n_f \\ card(\Pi_1 \bowtie^n \Pi_2), & \text{if } \Pi = \Pi_1 \bowtie^n \Pi_2 \end{cases} \tag{F.10}$$

To generalize Equation F.9 such that we can compute the cardinality of any join $\Pi = \Pi_1 \bowtie^n \Pi_2$ (e.g., including joins between a BGP with multiple star patterns and a star pattern), we consider two cases: (1) where Π_2 is a union $\Pi_2 = \Pi'_2 \cup \Pi''_2$, and (2) where Π_2 is a selection $\Pi_2 = [[P]]^{n_1}_f$. The cardinality of the join can thus be estimated using the following formula:

$$card(\Pi_1 \bowtie^n \Pi_2) = \begin{cases} card(\Pi_1 \bowtie^n \Pi'_2) + \\ \qquad card(\Pi_1 \bowtie^n \Pi''_2), & \text{if } \Pi_2 = \Pi'_2 \cup \Pi''_2 \\ card^{\bowtie}(\Pi_1, P, f), & \text{if } \Pi_2 = [[P]]^{n_1}_f \end{cases} \tag{F.11}$$

The function $card^{\bowtie}(\Pi, P, f)$ in the second case of Equation F.11 computes the cardinality of the join for a particular selection on the right side of the join, $[[P]]_f$. To achieve this estimation, we consider the estimated cardinality of Π and the selectivity of the join similar to Equation F.7. To avoid a significant overestimation due to the possible correlation between multiple join variables in the same join, we only consider the most selective join variable for any specific join. Recall the $\mathcal{B}(\mathcal{B}^S, P, v)$ function that returns the partitioned bitvector in $\mathcal{B}^S$ that corresponds to v's position in P, and let $S(\Pi, P)$ denote the set of star patterns in Π that join with P and $F(\Pi, f)$ denote the set of fragments in Π that join with f. For instance, for the execution plan in Figure F.12d and the compatibility graph in Figure F.5g, $S(\Pi, P_3) = \{P_1\}$ and $F(\Pi, f_5) = \{f_1, f_2\}$. Furthermore, given two star patterns P_1 and P_2, let $v(P_1, P_2) = \{v \mid v \in vars(P_1) \cap vars(P_2)\}$, i.e., the set of join variables. The cardinality of the join between a plan Π and a selection $[[P]]_f$ is, given the `DISTINCT` keyword, generalized from Equation F.7 as follows:

$$card^{\bowtie}_D(\Pi, P, f) = card(\Pi) \cdot \min_{P' \in S(\Pi,P) \wedge v \in v(P,P')} \frac{\sum_{f' \in F(\Pi,f)} card^P(\mathcal{B}(\mathcal{B}^S(f), P, v) \cap \mathcal{B}(\mathcal{B}^S(f'), P', v))}{\sum_{f' \in F(\Pi,f)} card^P(\mathcal{B}(\mathcal{B}^S(f'), P', v))} \tag{F.12}$$

As an example, consider computing the cardinality $card(\Pi)$ of the plan Π visualized in Figure F.12d using the `DISTINCT` keyword. Since Π is a union, we compute the cardinality of $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$ and $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$ and let $card(\Pi) = card(\Pi_1) + card(\Pi_2)$. Using

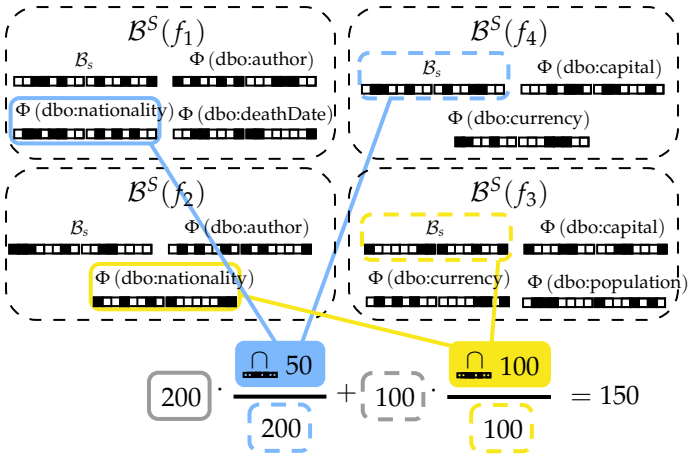


Fig. F.10: Estimating the cardinality of $\Pi = ([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$ with the `DISTINCT` keyword using the cardinalities from Table F.1 and Equation F.12.

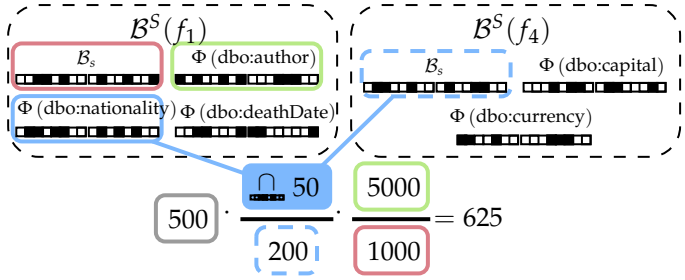
Equation F.12 on Π_1 and Π_2, we get the formula $card(\Pi) = 200 \cdot (50/200) + 100 \cdot (100/100) = 150$ as visualized in Figure F.10 (the gray values are the cardinalities of the left selections in each join obtained using Equation F.4).

For queries without the `DISTINCT` keyword, we once again consider the average predicate occurences. However, since the predicate occurrences in Π are already considered in $card(\Pi)$ in Equation F.12, we only consider the average number of occurrences in f for each triple pattern in P that does not join with Π on the object. The cardinality of the join between a plan Π and selection $[[P]]_f$, without the `DISTINCT` keyword, is computed as:

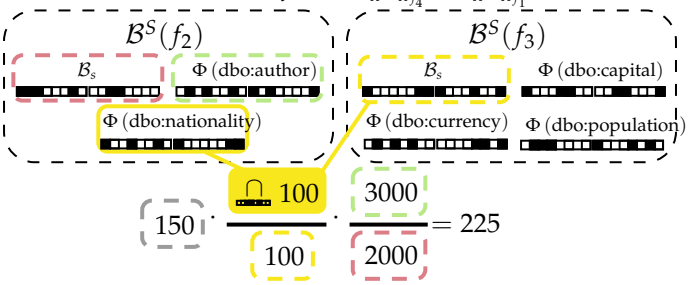
$$card^{\bowtie}_S(\Pi, P, f) = card^{\bowtie}_D(\Pi, P, f) \cdot \prod_{p \in preds(P): (s,p,o) \in P \wedge o \notin v(P,P') \forall P' \in S(\Pi,P)} \left(\frac{card^P(\mathcal{B}^S(f).\Phi(p))}{card^P(\mathcal{B}^S(f).\mathcal{B}_s)} \right) \quad \text{(F.13)}$$

Once again, computing the cardinality of Π in Figure F.12d not including the `DISTINCT` keyword is $card(\Pi) = card(\Pi_1) + card(\Pi_2)$. Using Equation F.13 on each of these yields the equation $card(\Pi) = 500 \cdot (50/200) \cdot (5000/1000) + 150 \cdot (100/100) \cdot (3000/2000) = 625 + 225 = 850$. Figure F.11 visualizes this computation.

Using the cardinality estimation shown in Equation F.10, Algorithm 13 shows how the transfer cost of a query execution plan Π on a node n is computed taking into account the locality of the fragments. First, if $\Pi = [[P]]^{n_i}_f$, i.e., Π is a selection, the algorithm checks whether $n = n_i$ (line 4); if they are equal it is 0 (since it incurs no transfer cost), otherwise the transfer cost of



(a) Cardinality of $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$



(b) Cardinality of $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$

Fig. F.11: Estimating the cardinality of Π in Figure F.12d without the `DISTINCT` modifier for (a) $\Pi_1 = [[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}$ and (b) $\Pi_2 = [[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2}$. The output of Equation F.10 is thus the sum of the two formulas ($625 + 225 = 850$).

Π is equal to the cardinality of the selection (Equation F.5). For instance, the transfer cost of the execution plan shown in Figure F.12c ($[[P_3]]^{n_1}_{f_5}$) on n_1 is 0 since f_5 is available on n_1.

If, instead, $\Pi = \Pi_1 \cup \Pi_2$, i.e., Π is a union, the transfer cost is the sum of the transfer costs for Π_1 and Π_2 (line 6). For instance, the transfer cost of the execution plan shown in Figure F.12a ($[[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}$) on n_1 is $5000 + 3000 = 8000$, since neither f_1 or f_2 is available on n_1.

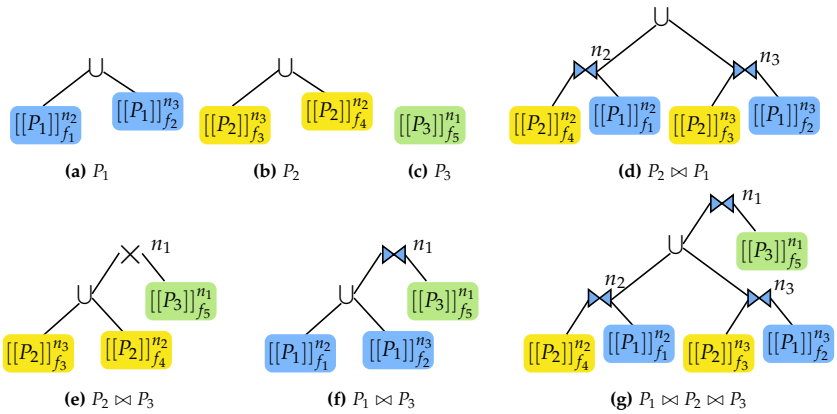
Otherwise, if $\Pi = \Pi_1 \times^{n_i} \Pi_2$, i.e., Π is a Cartesian product, the transfer cost is the sum of the transfer costs for Π_1 and Π_2 (line 6), plus the cardinality of the Cartesian product if it is delegated to a different node than the one processing the (sub)plan, i.e., if $n \neq n_i$ (since they have to be transferred from n_i to n).

Finally, if $\Pi = \Pi_1 \bowtie^{n_i} \Pi_2$, i.e., Π is a join, we once again take advantage of the fact that the right side of a join is always either a selection or a union of selections; in the latter case, we aggregate the transfer cost over each subplan in the union (line 12). However, if the right side of the join is a selection $\Pi_2 = [[P]]^{n_j}_f$, we start by estimating the transfer cost of the left side of the join (line 14); if $n_j \neq n_i$, we further add in line 15 the cardinality of the join

Algorithm 13 Compute the transfer cost of a query execution plan

Input: A query execution plan Π; a node n
Output: The estimated transfer cost *cost*

```
1:  function transferCost(Π,n)
2:      cost ← 0;
3:      if Π = [[P]]_f^{n_i} then
4:          if n ≠ n_i then cost ← card(P, f);
5:      else if Π = Π_1 ∪ Π_2 then
6:          cost ← transferCost(Π_1, n) + transferCost(Π_2, n);
7:      else if Π = Π_1 ×^{n_i} Π_2 then
8:          cost ← transferCost(Π_1, n_i) + transferCost(Π_2, n_i);
9:          if n_i ≠ n then cost ← cost + card(Π);
10:     else if Π = Π_1 ⋈^{n_i} Π_2 then
11:         if Π_2 = Π'_2 ∪ Π''_2 then
12:             cost ← transferCost(Π_1 ⋈^{n_i} Π'_2, n) + transferCost(Π_1 ⋈^{n_i}
    Π''_2, n);
13:         else if Π_2 = [[P]]_f^{n_j} then
14:             cost ← transferCost(Π_1, n_i);
15:             if n_i ≠ n_j then cost ← cost + card_S^⋈(Π_1, P, f);
16:         if n ≠ n_i then cost ← cost + card(Π);
17:     return cost;
```



(a) P_1 **(b)** P_2 **(c)** P_3 **(d)** $P_2 \bowtie P_1$

(e) $P_2 \bowtie P_3$ **(f)** $P_1 \bowtie P_3$ **(g)** $P_1 \bowtie P_2 \bowtie P_3$

Fig. F.12: Best query execution plan for each subquery in the DP table (Table F.2).

(since these results should have to be sent back to n_i). Furthermore, if $n_i \neq n$, we add in line 16 the cardinality of the execution plan to the cost, since the results have to be transferred from n_i to n.

Given the transfer cost in Algorithm 13, the cost of processing a query

execution plan Π on a node n is the transfer cost plus the cardinality of Π. This is formally defined as follows:

$$cost_n(\Pi) = transferCost(\Pi, n) + card(\Pi) \tag{F.14}$$

Given the cost function in Equation F.14, we compute the cost of each possible join delegation and apply Dynamic Programming (DP) to achieve the execution plan with the lowest cost. Table F.2 shows the best execution plan in the DP table for each (sub)plan for processing query Q (Figure F.3a) on node n_1 in the running example; Figure F.12 visualizes the execution plans in Table F.2.

Table F.2: Entries in the DP table for query Q (Figure F.3a)

Subquery	Execution Plan	Cardinality	Cost
P_1	$[[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}$	8,000	8,000
P_2	$[[P_2]]^{n_3}_{f_3} \cup [[P_2]]^{n_2}_{f_4}$	650	650
P_3	$[[P_3]]^{n_1}_{f_5}$	9,000	9,000
$P_2 \bowtie P_1$	$([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})$	**850**	1,700
$P_2 \bowtie P_3$	$([[P_2]]^{n_3}_{f_3} \cup [[P_2]]^{n_2}_{f_4}) \times^{n_1} [[P_3]]^{n_1}_{f_5}$	5,850,000	5,850,650
$P_1 \bowtie P_3$	$([[P_1]]^{n_2}_{f_1} \cup [[P_1]]^{n_3}_{f_2}) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$	1,688	9,688
$P_2 \bowtie P_1 \bowtie P_3$	$(([[P_2]]^{n_2}_{f_4} \bowtie^{n_2} [[P_1]]^{n_2}_{f_1}) \cup ([[P_2]]^{n_3}_{f_3} \bowtie^{n_3} [[P_1]]^{n_3}_{f_2})) \bowtie^{n_1} [[P_3]]^{n_1}_{f_5}$	154	1,004

6 Query Execution

Until now, we have described in Section 5 how Lothbrok obtains a query execution plan using compatibility graphs and locality information provided by the SPBF indexes. In this section, we detail how Lothbrok evaluates a query given a query execution plan.

Given a BGP P, a compatibility graph $G^C = G^C(P, I^S)$, and a query execution plan Π over P and G^C, Lothbrok processes P by processing the operations specified in Π and, in doing so, delegating joins and Cartesian products to the nodes specified in Π. The intermediate results from previous steps are used as input to subqueries at a later stage in the query execution plan. In case of a distributed join, the intermediate results are transferred along with the partial query to use local bind joins similar to [3, 30]. To formalize how star patterns in the query execution plan are processed over the fragments, we define a so-called *selector* function in line with related work [3, 6, 30]. The selector function returns the results of processing the star pattern over a fragment given a set of solution mappings, i.e., the set of stars in the fragment that constitute the answer to the star pattern, as follows:

Definition F.18 (Selector Function [3, 6, 30])
Given a node n, a star pattern P, and a finite set of distinct solution mappings Ω, the star pattern-based selector function for P and Ω, denoted $s_{(P,\Omega)}$ is for every fragment f in n's local datastore defined as follows.

$$s_{(P,\Omega)}(f) = \begin{cases} \{t \in T \mid T \subseteq f \wedge T[P]\} & \text{if } \Omega = \emptyset \\ \{t \in T \mid T \subseteq f \wedge \exists \mu \in [[P]]_f, & \\ \quad \mu' \in \Omega : \mu[P] = T \wedge \mu' \subseteq \mu\} & \text{otherwise.} \end{cases}$$

In line with [3, 6, 30], and to avoid long-running requests on each node, we apply pagination to the results of star pattern requests, i.e., we group the results into reasonably sized pages to avoid excessive data transfer. The page size used in our experimental evaluation (Section 7) is the page size recommended by related work [3, 6, 30], i.e., 100. However, for ease of presentation, we assume that all results can fit into one page when presenting the approach to query processing. Furthermore, to avoid underestimating costs caused by the selector function returning some duplicate values (e.g., when the same subject has multiple object values for a specific predicate), our implementation always uses $card_S$ (Equation F.5) and $card_S^{\bowtie}$ (Equation F.13) for cardinality estimations, regardless of whether or not the `DISTINCT` keyword is given. Last, given a star pattern P, a node n, a fragment f_i, and a finite set of solution mappings Ω, $sel_n(f_i, P, \Omega)$ denotes the result of invoking $s_{(P,\Omega)}(f_i)$ on n.

Let I_n^S denote a node n's SPBF index. The `evaluatePlan` function in Algorithm 14 defines a recursive function that processes a query execution plan on a node n by using the selector function defined in Definition 33 for selections in the plan and making recursive calls to the nodes specified in the plan.

Consider, for instance, the query execution plan Π shown in Figure F.12g for query Q in Figure F.3a processed by node n_1 in the running example. Figure F.13 shows an overview of which parts of the query are sent to which node during query processing. Since Π is of type join, the function enters the if statement in line 6. Here, the function first makes a recursive call (since the join was delegated to node n_1) with the left-most subplan, i.e., $\Pi_1 = ([[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}) \cup ([[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2})$ (visualized in Figure F.12d), in line 7.

Since Π_1 is of type union, Algorithm 14 in lines 10-11 makes two recursive calls for the two subplans $\Pi_1 = [[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}$ and $\Pi_2 = [[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2}$. Note that these two recursive calls can be processed concurrently and indeed is done so in the implementation of Lothbrok. This step is shown in Figure F.13a where Π_1 is sent to node n_2 and Π_2 is sent to node n_3. Since both subplans follow the same structure, and thus the same evaluation process, we will only explain what happens when processing Π_1.

When processing the plan Π_1 from above, Algorithm 14 first calls the

Algorithm 14 Evaluate a join plan

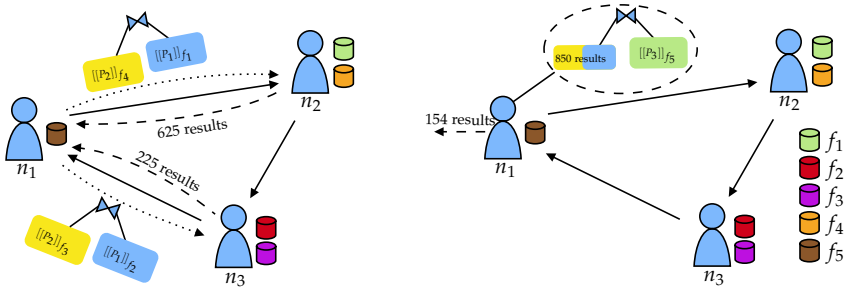
Input: A join plan Π; a node n; a set of solution mappings Ω
Output: A set of solution mappings Ω

```
1:  function evaluatePlan(Π,n,Ω = {∅})
2:      if Π = Π1 ⋈^{n_i} Π2 then
3:          Ω1 ← evaluatePlan(Π1, n_i, Ω);
4:          Ω2 ← evaluatePlan(Π2, n_i, Ω);
5:          Ω ← Ω1 × Ω2;
6:      else if Π = Π1 ⋈^{n_i} Π2 then
7:          Ω ← evaluatePlan(Π1, n_i, Ω);
8:          Ω ← evaluatePlan(Π2, n_i, Ω);
9:      else if Π = Π1 ∪ Π2 then
10:         Ω1 ← evaluatePlan(Π1, n, Ω);
11:         Ω2 ← evaluatePlan(Π2, n, Ω);
12:         Ω ← Ω1 ∪ Ω2;
13:     else if Π = [[P]]_f then
14:         N ← I_n^S.η(f);
15:         if n ∈ N then n_i ← n;
16:         else n_i ← takeOne(N);
17:         φ ← sel_{n_i}(f, P, Ω);
18:         Ω ← Ω ⋈ {μ | dom(μ) = vars(P) and μ[P] ∈ φ};
19:     return Ω;
```

`evaluatePlan` on node n_2 for the subplan $[[P_2]]_{f_4}$, i.e., the selection for P_2 over f_4, in line 7. The `takeOne` function in line 16 selects a random node with the fragment in its local datastore if the node that processes the subquery does not store the fragment locally. In this case, since n_2 stores f_4, it calls the selector function for P_2 over f_4 locally in line 17. The 500 results of processing P_2 over f_4 (cf. Table F.1) are then joined with the singleton set of bindings Ω that includes the empty mapping (i.e., a mapping compatible with any mapping) in line 18 and returned in line 19.

Upon receiving the 500 results in line 7, Algorithm 14 makes another recursive call in line 8 to `evaluatePlan` on node n_2 for the subplan $[[P_1]]_{f_1}$, i.e., the selection for P_1 over f_1 with the 500 intermediate results in Ω. Again, n_2 calls the local selector for P_1 over f_1 using the intermediate results in Ω as bindings. This results in 625 intermediate results in Ω that are the result of processing $P_1 \bowtie P_2$ over f_1 and f_4, which are returned by the function in line 19.

While n_2 found the 625 results from processing $[[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}$ in the recursive call in line 10, n_3 found the additional 225 results of processing $[[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2}$ in the recursive call in line 11 following the same steps



(a) Delegating subqueries to n_2 and n_3 **(b)** Processing $[[P_3]]_{f_5}$ locally on node n_1

Fig. F.13: Processing Π in Figure F.12g on n_1 by (a) delegating $[[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}$ to n_2 and $[[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2}$ to n_3 concurrently and (b) processing the join between these 850 results and $[[P_3]]_{f_5}$ locally on n_1 to achieve the 154 results (solid arrows denote neighbors, dotted arrows subquery delegation, and dashed arrows transferring of intermediate results). n_1 can send intermediate results to n_3 since it is within its horizon.

as described above for n_2. In line 12, these results are combined and 850 bindings are returned in line 19, which is visualized on Figure F.13a as n_2 returning 625 results to n_1 and n_3 returning 225 results to n_1.

The 850 intermediate results in Ω found by processing $([[P_2]]_{f_4} \bowtie^{n_2} [[P_1]]_{f_1}) \cup ([[P_2]]_{f_3} \bowtie^{n_3} [[P_1]]_{f_2})$ in line 7 are used as bindings for the recursive call made in line 8 for the subplan $[[P_3]]_{f_5}$. This is visualized in Figure F.13b. Since n_1 stores f_5 locally, it calls the local selector for P_3 over f_5 and Ω in line 17. The 154 results of processing P_3 over f_5 are joined with Ω in line 18 and returned as the final results in line 19.

As mentioned above, our implementation uses pagination of the results meaning, for instance, when processing the subplan $[[P_2]]_{f_4}$ in line 7, the 500 results would be split into multiple pages. In the implementation of Lothbrok, nodes at subsequent steps in the pipeline start processing joins as soon as they receive some intermediate bindings. For instance, in the running example, n_1 starts processing the join between $P_2 \bowtie P_1 \bowtie P_3$ locally as soon as it receives results for $P_2 \bowtie P_1$ from either n_2 or n_3.

7 Experimental Evaluation

The experimental evaluation compares Lothbrok with two state-of-the-art approaches building on P2P systems: Piqnic [4] and ColChain [6] with the query optimization approach outlined in [5]. To do this, we implemented the fragmentation, indexing, and cardinality estimation approach as a separate package in Java 8 and modified Piqnic's and ColChain's query processors to use it. Like ColChain and Piqnic, Lothbrok's query processor is imple-

mented as an extension to Apache Jena[2]. Fragments in our implementation are stored as HDT files [22], allowing for efficient processing of the star patterns. We provide all source code, experimental setup (queries, datasets, etc.), and the full experimental results on our website[3].

7.1 Experimental Setup

In this section, we detail the experimental setup, including a characterization of the used datasets and queries, the hardware and software setup, experimental configuration, as well as the evaluation metrics.

Datasets and Queries. To test the scalability of the approaches when the network is under heavy load, and to assess the impact of the query pattern on performance and network usage, we ran experiments with the synthetic WatDiv [9] benchmark using different dataset sizes: 10 million triples to 1 billion triples. Furthermore, to test Lothbrok in a realistic setting where users would upload several interlinked datasets to a network, and ask queries with varying complexity, we ran experiments using a well-known benchmark suite for federated RDF engines called LargeRDFBench [59]. LargeRDFBench comprises 13 different, interlinked datasets with over a billion triples in total. To provide a fair comparison between the systems with and without Lothbrok, we created an equal number of fragments for both fragmentations: characteristic sets (Section 4.2) and predicate-based. To do this, we iteratively merged the characteristic set fragments with the fewest number of subjects into larger fragments following the approach outlined in Section 4.2 until the number of fragments equalled the number of predicate-based fragments. The characteristics of the datasets are shown in Table F.3. Furthermore, to assess the impact of reducing the number of characteristic sets on query completeness, we ran similar experiments where we did not create an equal number of fragments for Lothbrok, i.e., where we created one fragment for each characteristic set that describes at least 50 subjects and provide the results on our website[3]; since these results are quite similar to the ones presented in this section, we will not report on them further.

LargeRDFBench includes 40 different queries [59] that are divided into five different categories of varying complexity and result set sizes: Simple (S), Complex (C), Large Data (L), and Complex and High Data Sources (CH).

For WatDiv, we used WatDiv *star query loads* from [3] consisting of 1-3 star patterns, called the `watdiv-1_star`, `watdiv-2_star`, and `watdiv-3_star` query loads, as well as a query load consisting of path queries, i.e., queries where each star pattern only has one triple pattern, called the `watdiv_path` query load. Each of these query loads consists of 6,400 different queries. Furthermore, we combine the aforementioned query loads into a single

[2] https://jena.apache.org

[3] https://relweb.cs.aau.dk/lothbrok

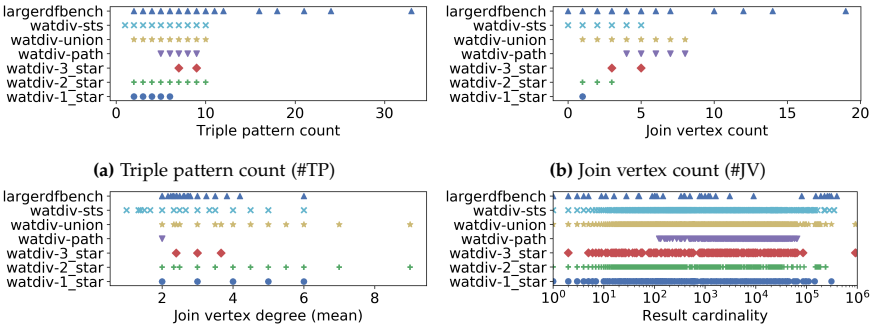
Table F.3: Characteristics of the used datasets

Dataset	#triples	#subjects	#predicates	#objects
`LargeRDFBench`	1,003,960,176	165,785,212	2,160	326,209,517
LinkedTCGA-M	415,030,327	83,006,609	6	166,106,744
LinkedTCGA-E	344,576,146	57,429,904	7	84,403,402
LinkedTCGA-A	35,329,868	5,782,962	383	8,329,393
ChEBI	4,772,706	50,477	28	772,138
DBPedia-Subset	42,849,609	9,495,865	1,063	13,620,028
DrugBank	517,023	19,693	119	276,142
GeoNames	107,950,085	7,479,714	26	35,799,392
Jamendo	1,049,647	335,925	26	440,686
KEGG	1,090,830	34,260	21	939,258
LinkedMDB	6,147,996	694,400	222	2,052,959
NYT	335,198	21,666	36	191,538
SWDF	103,595	11,974	118	37,547
Affymetrix	44,207,146	1,421,763	105	13,240,270
`watdiv10M`	10,916,457	521,585	86	1,005,832
`watdiv100M`	108,997,714	5,212,385	86	9,753,266
`watdiv1000M`	1,092,155,948	52,120,385	86	92,220,397

query load called `watdiv-union`. Last, we created a query load with 19,968 queries from the WatDiv stress testing query templates (156 per node) called `watdiv-sts`. The complete set of queries is available on our website[3]. Figure F.14 shows an overview of the following characteristics of each load [3, 10]: Triple pattern count #TP (Figure F.14a), join vertex count #JV (Figure F.14b), join vertex degree DEG (Figure F.14c), result cardinality #Results (Figure F.14d), mean triple pattern selectivity $\text{SEL}_{\mathcal{G}}(tp)$ (Figure F.14e), and join vertex type (Figure F.14f).

Experimental Configuration. We compare the following systems: (1) PIQNIC [4] using PPBF indexes [5] (PIQNIC), (2) LOTHBROK on top of PIQNIC (LOTHBROK $_{\text{PIQNIC}}$), (3) COLCHAIN [6] using PPBF indexes (COLCHAIN), and (4) LOTHBROK on top of COLCHAIN (LOTHBROK $_{\text{COLCHAIN}}$). All configurations were run on networks with 128 nodes. To assess the scalability of LOTHBROK under load, we ran 156 `watdiv-sts` queries concurrently on each node over 8 different configurations where 2^i nodes issue queries concurrently such that $0 \leq i \leq 7$ (i.e., up to all 128 nodes). Furthermore, to analyze the impact of the query pattern on performance, we ran the WatDiv star query loads over each WatDiv dataset size such that for each star query load, each node issued 50 queries. Lastly, we tested the performance of LOTHBROK over each individual query in LargeRDFBench by running the queries sequentially in random order on three randomly selected nodes and report the average result.

Hardware Configuration. For all configurations and P2P systems, we ran 128 nodes concurrently on a virtual machine (VM) with 128 vCPU cores with



(a) Triple pattern count (#TP)

(b) Join vertex count (#JV)

(c) Join vertex degree (DEG)

(d) Result cardinality (#Results)

Query load	SS	SO	OO
watdiv-1_star	100%	0%	0%
watdiv-2_stars	55.38%	34.31%	10.31%
watdiv-3_stars	57.67%	30.77%	11.53%
watdiv-paths	0%	100%	0%
watdiv-union	53.27%	41.27%	5.46%
watdiv-sts	55.57%	35.83%	8.6%
largerdfbench	41.41%	45.81%	12.78%

(e) Mean TP selectivity ($\text{SEL}_{\mathcal{G}}(tp)$)

(f) Join vertex type over each query load

Fig. F.14: Characteristics of all query loads (WatDiv query loads over `watdiv100M`; statistics over the `watdiv10M` and `watdiv100M` datasets can be found on our website[3]).

a clock speed of 2.5GHz, 64KB L1 cache, 512KB L2 cache, 8192KB L3, and a total of 2TB main memory. To spread out resources evenly across nodes, all nodes were restricted to use 1 vCPU core and 15GB memory, enforced using the `-Xmx` and `-XX:ActiveProcessorCount` options for the JVM. Furthermore, to simulate a more realistic scenario, where nodes are not run on the same machine, we simulated a connection speed of 20 MB/s.

Evaluation Metrics. We used measured the following metrics:

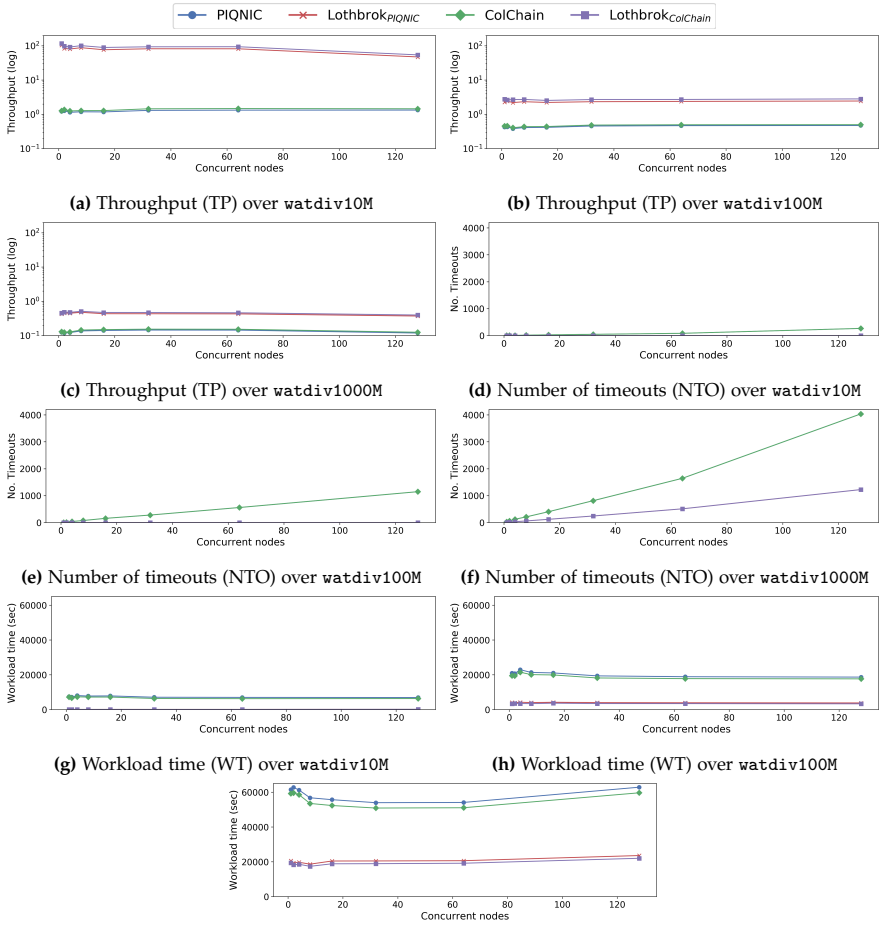
- *Workload Time (WT):* The amount of time (in milliseconds) it takes to complete an entire workload including queries that time out.
- *Throughput (TP):* The number of completed queries in the workload divided by the total workload time (i.e., number of queries per minute).
- *Number of Timeouts (NTO):* The number of queries that timed out (timeout being 1200 seconds).
- *Query Execution Time (QET):* The amount of time (in milliseconds) elapsed between when a query is issued and when its processing has finished.
- *Query Response Time (QRT):* The amount of time (in milliseconds) elapsed between when a query is issued and when the first result is computed.
- *Query Optimization Time (QOT):* The amount of time (in miliseconds) elapsed between when a query is issued and when the optimizer has

finished (i.e., when query execution starts).
- *Number of Requests (REQ):* The number of requests made between nodes when processing a query (including requests made from nodes that have been delegated subqueries).
- *Number of Transferred Bytes (NTB):* The amount of data (in bytes) transferred between nodes when processing a query (including data transferred to and from nodes that have been delegated subqueries).
- *Number of Relevant Nodes (NRN):* The number of distinct nodes that replicate fragments containing relevant data to a query.
- *Number of Relevant Fragments (NRF):* The number of distinct fragments containing relevant data to a query.

Software Configuration. Unless otherwise specified, we used the following parameters when running the systems. For ColChain, we used the following parameters recommended in [6]: Community Size: 20, Number of Communities: 200. For Piqnic, we use the following parameters recommended in [4]: Time-to-Live (number of hops): 5, Number of Neighbors: 5. The replication factor for Piqnic (i.e., the percentage of nodes replicating each fragment) was matched with the size of the communities in ColChain to provide a better comparison. Nodes were randomly assigned neighbors throughout the network. The page size (i.e., how many results can be returned with each request, was set to 100. Furthermore, to limit the size of HTTP requests, the number of results that each system was allowed to attach to each request (i.e., $|\Omega|$ in Section 6) was set to $|\Omega| = 30$ The timeout for all systems and queries was set to 20 minutes (1,200 seconds).

7.2 Scalability under Load

In these experiments, we ran the `watdiv-sts` queries over each WatDiv dataset in configurations where 2^i nodes issued 156 queries from the `watdiv-sts` query load concurrently such that $0 \leq i \leq 7$. Figures F.15a-F.15c show the throughput (TP) of the `watdiv-sts` query load over each configuration in the scalability tests for the `watdiv10M` (Figure F.15a), `watdiv100M` (Figure F.15b), and `watdiv1000M` (Figure F.15c) datasets in logarithmic scale. Clearly, Lothbrok has a significantly higher throughput across all datasets and configurations compared to the approaches that do not include Lothbrok (i.e., Piqnic and ColChain). In fact, for `watdiv10M`, this increase in throughput is close to two orders of magnitude. While the increase in throughput that Lothbrok provides is smaller for both `watdiv100M` and `watdiv1000M`, Lothbrok still increases the throughput by close to an order of magnitude for these datasets. Furthermore, while some results show that ColChain has a slightly higher throughput than Piqnic, both with and without Lothbrok on top, this difference is relatively negligible. Last, the results



(a) Throughput (TP) over `watdiv10M`

(b) Throughput (TP) over `watdiv100M`

(c) Throughput (TP) over `watdiv1000M`

(d) Number of timeouts (NTO) over `watdiv10M`

(e) Number of timeouts (NTO) over `watdiv100M`

(f) Number of timeouts (NTO) over `watdiv1000M`

(g) Workload time (WT) over `watdiv10M`

(h) Workload time (WT) over `watdiv100M`

(i) Workload time (WT) over `watdiv1000M`

Fig. F.15: Throughput (TP), number of timeouts (NTO), and workload time (WT) for `watdiv-sts` over the `watdiv10M`, `watdiv100M`, and `watdiv1000M` datasets.

show that the throughput of Lothbrok is relatively stable when increasing numbers of nodes issue queries concurrently. In fact, even when every node in the network issue queries concurrently, the throughput is relatively close to the highest throughput throughout the configurations.

Figures F.15d-F.15f show the number of queries that timed out (TO) of the `watdiv-sts` query load over each configuration for each WatDiv dataset. As expected, the number of timeouts increases relatively linearly with the number of nodes issuing queries concurrently. This is due to the fact that when more nodes issue queries, more queries in total are executed, meaning the total number of the queries that time out increases. Generally, the queries

that time out correspond to query templates that result in a large number of intermediate results, e.g., by using the `owl:sameAs` predicate. Furthermore, Pıqnic and ColChain incur significantly more timeouts without Lothbrok compared to with Lothbrok. In fact, for both `watdiv10M` and `watdiv100M`, Lothbrok experiences no timeouts while Pıqnic and ColChain experience 267 timeouts for `watdiv10M` and 1,148 timeouts for `watdiv1000M`. Even for `watdiv1000M`, the number of timeouts experienced by Lothbrok is just 1,151 while Pıqnic and ColChain both experience 4,036 timeouts. Furthermore, Pıqnic and ColChain incur the exact same number of timeouts.

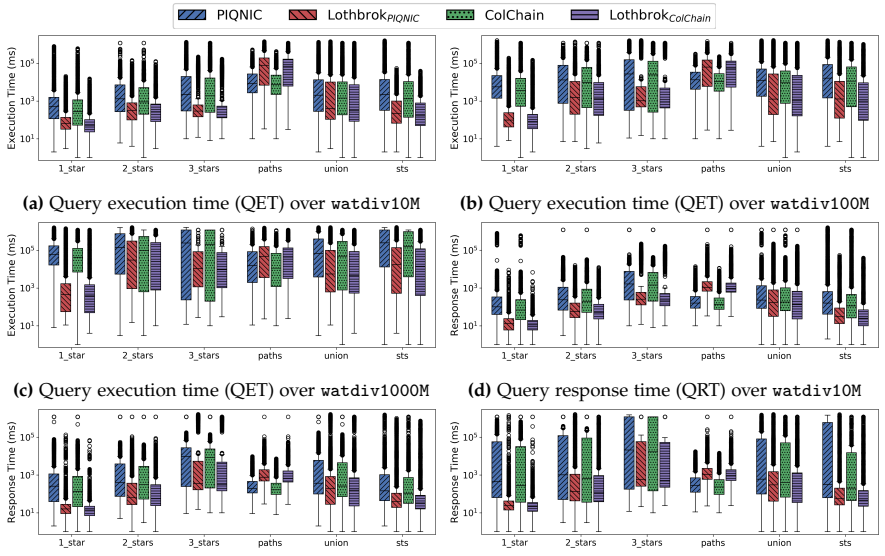
Figures F.15g-F.15i show the workload time (WT) for each configuration. In line with the throughput and number of timeouts, Lothbrok incurs a significantly lower average workload time than Pıqnic and ColChain across all experiments and datasets. The slight decrease in the workload time for fewer nodes can be attributed to the network being able to process more queries concurrently when the overall load is relatively low. Nevertheless, the average workload time only increases slightly even when all nodes issue queries concurrently.

Overall, our experimental results show that, even when the network is under heavy query processing load, Lothbrok increases the query throughput and decreases the average workload time significantly compared to state-of-the-art decentralized systems. In fact, the increase in performance is up to two orders of magnitude. As a result, Lothbrok is also able to finish more queries without timing out.

7.3 Impact of Query Pattern

To test the impact of the query pattern on the performance of Lothbrok, we ran the `watdiv-1_star`, `watdiv-2_star`, `watdiv-3_star`, `watdiv-path`, `watdiv-union`, and `watdiv-sts` query loads on each system; the `watdiv-sts` queries consist of, on average, more selective star patterns compared to the other WatDiv query loads (Figure F.14).

Figures F.16a-F.16c show the execution time (QET) for each WatDiv query load over each WatDiv dataset, and Figures F.16d-F.16f show the response time (QRT) for each WatDiv query load in logarithmic scale. Our results show that Lothbrok has significantly better performance across all datasets for almost every query load. As expected, the improvement in performance is more significant for the query loads with a lower number of star patterns. This is due to the fact that since the star patterns within these queries represent a large part of the query, Lothbrok has to issue fewer requests overall, lowering the network overhead. For instance, the queries in the `watdiv-1_star` query load can by Lothbrok be answered by issuing 0.89



(a) Query execution time (QET) over `watdiv10M`

(b) Query execution time (QET) over `watdiv100M`

(c) Query execution time (QET) over `watdiv1000M`

(d) Query response time (QRT) over `watdiv10M`

(e) Query response time (QRT) over `watdiv100M`

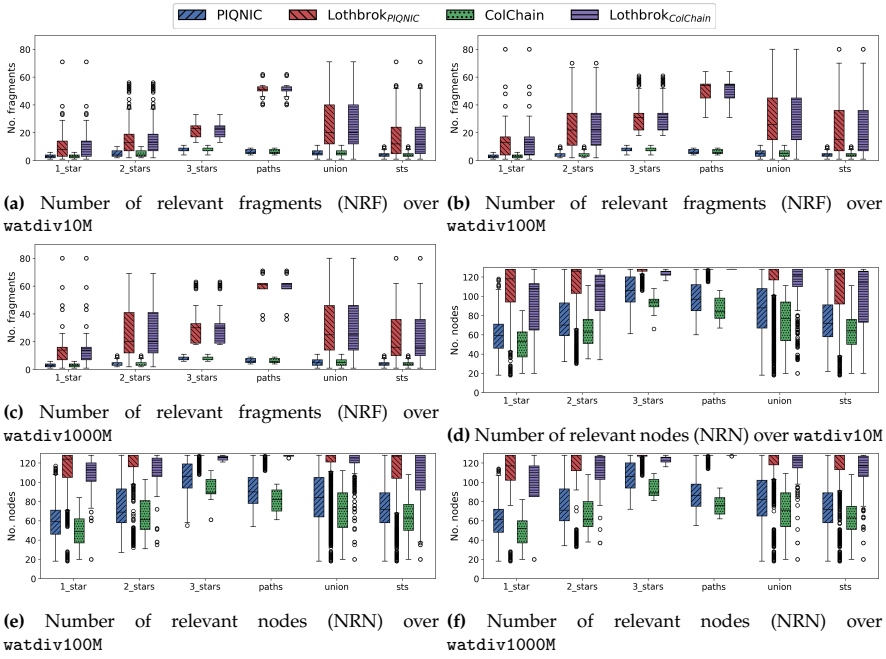
(f) Query response time (QRT) over `watdiv1000M`

Fig. F.16: Query execution time (QET) and query response time (QRT) for the WatDiv datasets and star queries.

requests per 90 results[4], whereas PIQNIC and COLCHAIN have to issue 9.27 requests per 90 results on average, for `watdiv1000M` in our experiments. In the `watdiv-3_star` query load, the improvement in performance is more modest across the datasets since each star pattern is a relatively small part of the query resulting in a higher number of requests; however, on average, we still see a performance increase of up to an order of magnitude.

We notice that for the `watdiv-path` query load, LOTHBROK actually has a slightly worse performance both in terms of QET and QRT compared to PIQNIC and COLCHAIN due to higher network usage. Figure F.17 shows the number of relevant fragments (NRF) and the number of relevant nodes (NRN) for each query load over each dataset after optimization (similar figures are provided for NRF and NRN before optimization on our website[3]). Analyzing these results, we see that the decreased performance for `watdiv-path` is caused by LOTHBROK having a significantly larger number of relevant fragments and by extension a larger number of relevant nodes compared to PIQNIC and COLCHAIN. In fact, this is the case for all the WatDiv query loads (9 times larger for `watdiv-path` while up to 5 times larger for the other query loads); however, for the other query loads, this is compensated by the increased performance that the query optimization approach provides.

[4]Even though one request can fetch up to 90 results, the average number of requests is lower than 1 since the nodes store some data locally.



(a) Number of relevant fragments (NRF) over `watdiv10M`

(b) Number of relevant fragments (NRF) over `watdiv100M`

(c) Number of relevant fragments (NRF) over `watdiv1000M`

(d) Number of relevant nodes (NRN) over `watdiv10M`

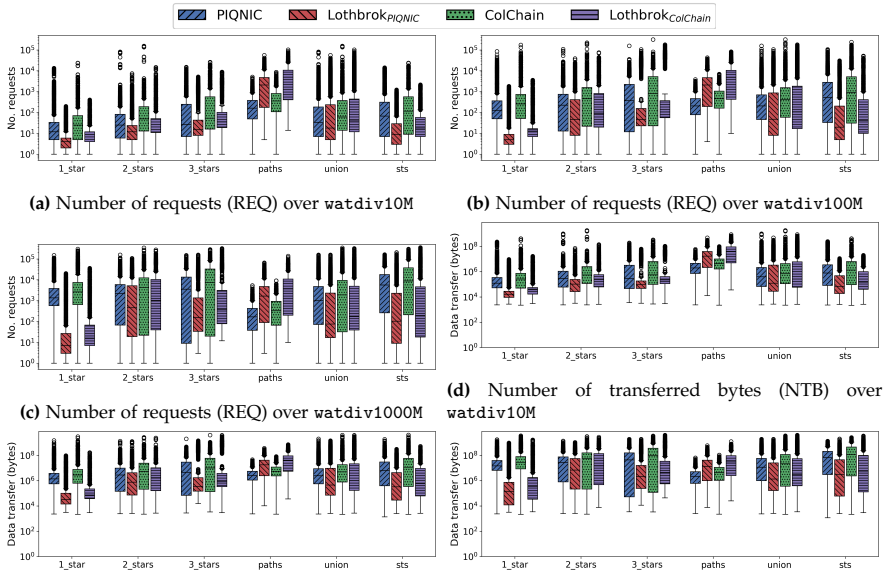
(e) Number of relevant nodes (NRN) over `watdiv100M`

(f) Number of relevant nodes (NRN) over `watdiv1000M`

Fig. F.17: Number of relevant fragments (NRF) and number of relevant nodes (NRN) for the WatDiv datasets and star queries.

This analysis is corroborated by the number of fragments pruned during optimization for each query load (figures provided on our website[3]); the `watdiv-path` query load has significantly less pruned fragments compared to the other query loads except `watdiv-1_star`. For PIQNIC and COLCHAIN, the number of relevant fragments will always equal the number of unique predicates in the query since one fragment is created per predicate; however, due to fragmenting the data based on characteristic sets, LOTHBROK can encounter multiple fragments for each unique predicate in the query. Furthermore, the number of relevant fragments is, on average, more than twice as high for LOTHBROK over the `watdiv-path` query load than over the other query loads. This is because most of the path queries use common predicates like `owl:sameAs`.

Nevertheless, the slightly worse performance for LOTHBROK over `watdiv-path` is compensated by the significantly improved performance over the other query loads, so we still see a performance increase for the `watdiv-union` query load. As such, our experimental results show that LOTHBROK is generally able to increase performance over queries with star-shaped subqueries (i.e., all other queries than path queries) significantly and that the increase in performance depends on the shape of the query; queries with



(a) Number of requests (REQ) over `watdiv10M`

(b) Number of requests (REQ) over `watdiv100M`

(c) Number of requests (REQ) over `watdiv1000M`

(d) Number of transferred bytes (NTB) over `watdiv10M`

(e) Number of transferred bytes (NTB) over `watdiv100M`

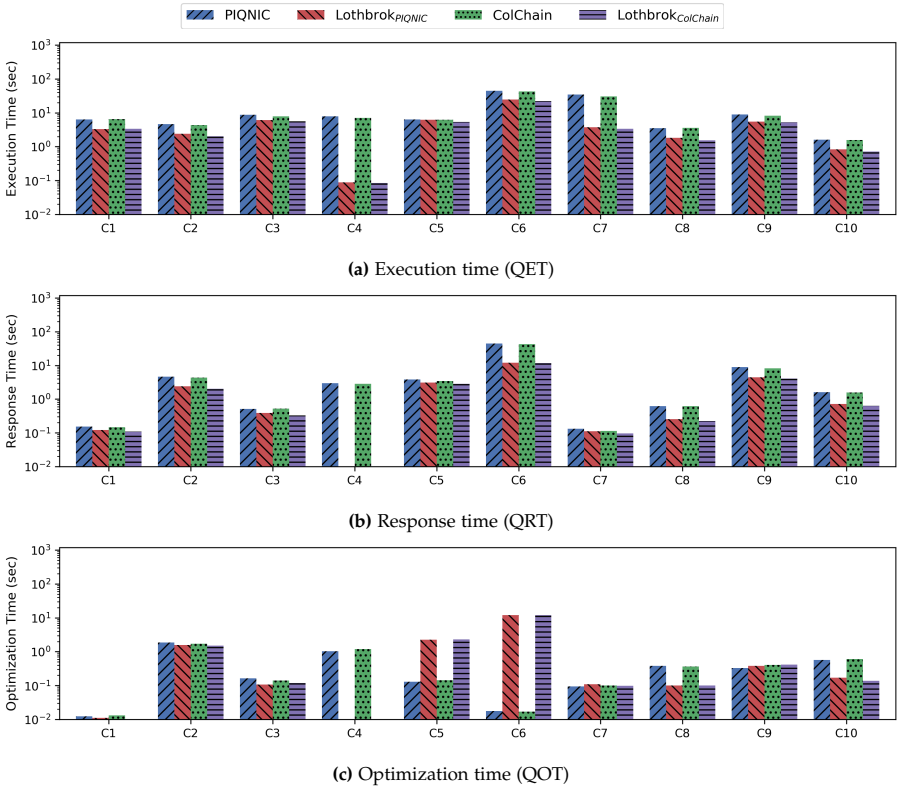
(f) Number of transferred bytes (NTB) over `watdiv1000M`

Fig. F.18: Number of requests (REQ) and number of transferred bytes (NTB) for the WatDiv datasets and star queries.

fewer but larger star patterns (cf. Figure F.14c) show a bigger performance increase than queries with many but small star patterns.

7.4 Network Usage

Figure F.18 shows the network usage when processing WatDiv queries over each WatDiv dataset in terms of the number of requests (Figures F.18a-F.18c) and the number of transferred bytes (Figures F.18d-F.18f) in logarithmic scale. Lothbrok incurs a significant lower network overhead for all query loads except `watdiv-path` despite the larger number of relevant fragments as discussed in Section 7.3. This is caused by Lothbrok having to send significantly fewer requests for each star pattern since a star pattern can be processed entirely over the relevant fragments, even if there are more fragments (and thus nodes) to send the requests to. Again, the query loads with a smaller number of star patterns see a larger decrease in network usage since larger parts of the queries can be processed by individual nodes. Since the queries in the `watdiv-path` query load do not benefit from the star pattern-based query processing, the network usage is slightly higher; however, even still, the `watdiv-union` shows an improvement in the network usage for Lothbrok. These results are in line with the experiments shown



(a) Execution time (QET)

(b) Response time (QRT)

(c) Optimization time (QOT)

Fig. F.19: Query execution time (a), response time (b), and optimization time (c) for the C query load over LargeRDFBench.

in Sections 7.2 and 7.3 and support the hypothesis that LOTHBROK increases performance by lowering the network overhead when processing queries, compared to state-of-the-art systems such as PIQNIC and COLCHAIN.

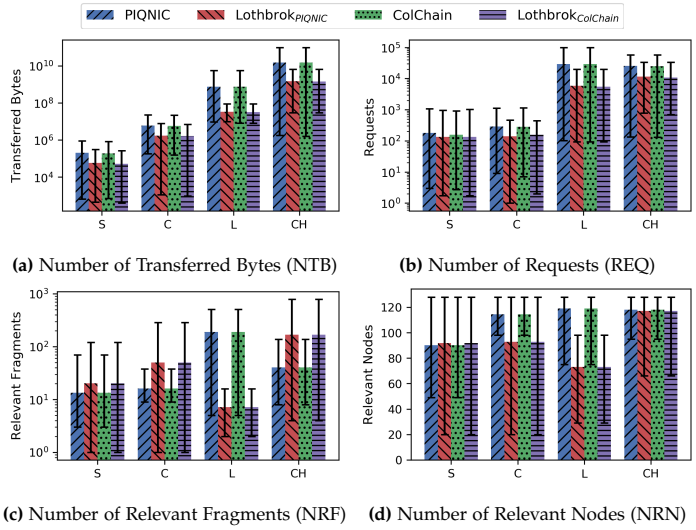
7.5 Performance of Individual Queries

In these experiments, we ran the LargeRDFBench queries three times on each system sequentially to test the performance of those individual queries and report the average results. Figure F.19 shows the execution time (Figure F.19a), response time (Figure F.19b), and optimization time (Figure F.19c) for the C query load over LargeRDFBench in logarithmic scale. Similar figures for the other LargeRDFBench query loads are provided on our website[3]. The results in Figure F.19 are similar to the remaining query loads; we show the C query load since this query load had the most diversity in the performance across the queries.

While, in our experiments, LOTHBROK provides an improvement for the execution time (Figure F.19a) across all the queries in LargeRDFBench, the improvement varies based on the query shape in line with the findings of [3, 13] and the query shape experiments shown in Section 7.3. For instance, query C4 consists of one highly selective star pattern with 6 unique predicates. LOTHBROK is thus able to answer C4 with one request to the only fragment with that predicate combination, while PIQNIC and COLCHAIN have to send at least one request per triple pattern. Hence, LOTHBROK has around two orders of magnitude better performance for this particular query. On the other hand, query C5 consists of four star patterns, two of which contain only one triple pattern with one of them being the very common `rdfs:label` predicate. As a result, LOTHBROK has more than twice the number of relevant fragments for C5 compared to both PIQNIC and COLCHAIN. Nevertheless, LOTHBROK still has slightly improved performance for C5 compared to PIQNIC and COLCHAIN since the query still contains two star patterns with three triple patterns each, meaning the increased optimization and communication overhead that the additional relevant fragments entail is offset by the benefits of processing the star patterns over the individual fragments. The response times (Figure F.19b) show a similar comparison between the systems as the execution times (Figure F.19a) with the exception of query C4. Again, the reason being that LOTHBROK can process this query with a single request, and therefore the first result is obtained immediately after receiving the response to the request.

However, the optimization times (Figure F.19c) differ quite significantly depending on the number of relevant fragments to the query. For instance, queries like C5 and C6 (that contain a star pattern consisting of a single triple pattern with a very common predicate) incur a significant number of relevant fragments for LOTHBROK (286 for C5 and 144 for C6) and thus a higher optimization time. This is the case, since a higher number of relevant fragments means a higher number of SPBFs have to be intersected which represents an overhead. In all of these cases, however, the benefits of processing entire star patterns over the fragments, in terms of decreased network overhead mean that the overall execution time is still lower for LOTHBROK. This is especially the case for C6, which contains a star pattern with 6 triple patterns that in PIQNIC and COLCHAIN have to be processed individually. On the other hand, queries like C4 that contain few very selective star patterns have a low optimization time for LOTHBROK, since each star pattern have very few relevant fragments. In the case of C4, PIQNIC and COLCHAIN have a relatively high number of relevant fragments due to one of the predicates being the common `owl:sameAs` predicate that occurs in multiple datasets. As a result, PIQNIC and COLCHAIN have a significantly higher optimization time for this query compared to LOTHBROK.

Figure F.20 shows the number of transferred bytes (Figure F.20a), the



(a) Number of Transferred Bytes (NTB) **(b)** Number of Requests (REQ)

(c) Number of Relevant Fragments (NRF) **(d)** Number of Relevant Nodes (NRN)

Fig. F.20: Number of Transferred Bytes (NTB) (a), Number of Requests (REQ) (b), Number of Relevant Fragments (NRF) (c), and Number of Relevant Nodes (NRN) (d) for each LArgeRDFBench query load.

number of requests (Figure F.20b), the number of relevant fragments (Figure F.20c), and the number of relevant nodes (Figure F.20d) for each LargeRDFBench query load in logarithmic scale. We provide figures displaying each measure in Figure F.20 for each individual LargeRDFBench query on our website[3]. As with the experiments shown in Section 7.4, Lothbrok clearly incurs a lower network usage than both Piqnic and ColChain, both in terms of data transfer (Figure F.20a) and the number of requests made (Figure F.20b). This, together with the performance experiments, shows that Lothbrok is able to reduce the network overhead significantly across all query loads and, in doing so, increase the performance overall.

Interestingly, while for most query loads, Lothbrok has a higher number of relevant fragments (Figure F.20c) in line with the experiments presented in Section 7.3, for the L query load, Lothbrok has a lower number of relevant fragments in most queries. The reason is that the queries in this query load mostly use data from the quite structured `linkedTCGA` datasets which contain few similar characteristic sets, thus incurring a low number of relevant fragments per star pattern. On the other hand, for Piqnic and ColChain, the fact that some star patterns with a low number of triple patterns include common predicates like `rdf:type` increases the number of relevant fragments. The number of relevant nodes (Figure F.20d) shows a similar trend to the number of relevant fragments since each fragment is replicated across 20 nodes; in some cases, however, where two relevant fragments are simultaneously repli-

cated by some of the same nodes, the actual number of relevant nodes will be a bit lower than when the relevant nodes replicate exactly one relevant fragment.

Our results are similar for all query loads (figures provided on our website[3]) and show that even for the complex queries in query loads C and CH and the queries with a large number of intermediate results in query load L, Lothbrok presents a significant performance increase because it lowers the communication overhead. For some queries, this is quite significant; for instance the queries C4 and S3 where Lothbrok increases execution time by up to two orders of magnitude. Furthermore, some queries in the L and CH groups that timed out for Piqnic and ColChain, such as L3 and CH2, finished within the timeout of 1200 seconds for Lothbrok. This is in line with the results presented in Section 7.2 and suggests that Lothbrok is able to complete more queries within the timeout than the state-of-the-art systems.

7.6 Summary

Our experimental evaluations show that Lothbrok significantly improves query performance while lowering the communication overhead compared to Piqnic and ColChain. Lothbrok does so by distributing subqueries to other nodes such that the estimated network cost is limited as much as possible, and by processing entire star patterns over the individual fragments. In doing so, Lothbrok decreases the network usage both in terms of the data transfer and number of requests, and increases performance by up to two orders of magnitude compared to the state of the art. Moreover, Lothbrok does so while providing scalable performance under load; in fact, even when all nodes in the network issue queries concurrently, Lothbrok maintains efficient query processing.

8 Conclusions

In this paper, we proposed Lothbrok a novel query optimization approach for SPARQL queries over decentralized knowledge graphs. Lothbrok builds upon recent work on decentralized Peer-to-Peer (P2P) systems [4, 6] and introduces a novel fragmentation technique based on characteristic sets [53], i.e., predicate families, as well as a novel indexing scheme that summarizes the sets of subjects and objects in a fragment using partitioned bitvectors. Furthermore, Lothbrok proposes a query optimization strategy based on cardinality estimation, fragment compatibility, and data locality that is able to delegate the processing of (sub)queries to other, neighboring nodes in the network that hold relevant data. We implemented our approach on top of two recent systems and evaluated Lothbrok's capabilities over well-known

benchmarking suites containing real-world data and queries, as well as the performance of Lothbrok under load using large-scale synthetic datasets and stress-testing query templates. The experimental results show that Lothbrok significantly reduces the network overhead when processing queries in a P2P network and, in doing so, increases performance by up to two orders of magnitude.

While we presented a novel distribution of the workload across nodes in a P2P network, Lothbrok also presents an opportunity to explore the effects of alternative strategies, e.g., for cost estimation, considering fragments optimized for object-object joins (Figure F.14f), or alternative fragmentation and allocation strategies, e.g., based on SHACL/ShEx shapes [57, 58]. Furthermore, we plan to expand the range of supported queries to include aggregation and analytical queries [24, 40] and to expand the framework with support of provenance both for data [11, 28, 31], so that the system has information about the origin of the data it uses, as well as for queries [34] so that the system can explain how query answers were computed.

References

[1] M. Acosta and M. Vidal, "Networks of linked data eddies: An adaptive web query processing engine for RDF data," in *ISWC 2015*. Springer, 2015, pp. 111–127.

[2] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "ANAPSID: an adaptive query processing engine for SPARQL endpoints," in *ISWC 2011*, 2011, pp. 18–34.

[3] C. Aebeloe, I. Keles, G. Montoya, and K. Hose, "Star pattern fragments: Accessing knowledge graphs through star patterns," *CoRR*, vol. abs/2002.09172, 2020. [Online]. Available: https://arxiv.org/abs/2002.09172

[4] C. Aebeloe, G. Montoya, and K. Hose, "A decentralized architecture for sharing and querying semantic data," in *ESWC 2019*, 2019, pp. 3–18.

[5] ——, "Decentralized indexing over a network of rdf peers," in *ISWC 2019*, 2019, pp. 3–20.

[6] ——, "ColChain: Collaborative linked data networks," in *WWW 2021*, 2021, pp. 1385–1396.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "Dbmss on a modern processor: Where does time go?" in *VLDB 1999*, 1999, pp. 266–277.

[8] A. Akhter, M. Saleem, A. Bigerl, and A.-C. Ngonga Ngomo, "Efficient rdf knowledge graph partitioning using querying workload," in *K-Cap 2021*, 11 2021.

[9] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified Stress Testing of RDF Data Management Systems," in *ISWC 2014*, 2014, pp. 197–212.

[10] ——, "Diversified stress testing of RDF data management systems," in *ISWC 2014*, 2014, pp. 197–212.

[11] A. B. Andersen, N. Gür, K. Hose, K. A. Jakobsen, and T. B. Pedersen, "Publishing Danish Agricultural Government Data as Semantic Web Data," in *JIST*, vol. 8943, 2014, pp. 178–186.

[12] C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?" in *ISWC 2013*, 2013, pp. 277–293.

[13] A. Azzam, C. Aebeloe, G. Montoya, I. Keles, A. Polleres, and K. Hose, "Wisekg: Balanced access to web knowledge graphs," in *WWW 2021*, 2021, pp. 1422–1434.

[14] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, and A. Polleres, "SMART-KG: hybrid shipping for SPARQL querying on the web," in *WWW 2020*, 2020, pp. 984–994.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[16] D. Brickley, R. V. Guha, and B. McBride, "Rdf schema 1.1," *W3C recommendation*, vol. 25, pp. 2004–2014, 2014.

[17] M. Cai and M. R. Frank, "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *WWW*, 2004, pp. 650–657.

[18] A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos, "Semagrow: optimizing federated SPARQL queries," in *SEMANTiCS 2015*, 2015, pp. 121–128.

[19] W. W. W. Consortium *et al.*, "Sparql 1.1 overview," 2013.

[20] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *ICDCS 2002*, 2002, pp. 23–32.

[21] M. Dumontier, A. Callahan, J. Cruz-Toledo, P. Ansell, V. Emonet, F. Belleau, and A. Droit, "Bio2rdf release 3: A larger, more connected network of linked data for the life sciences," in *ISWC 2014 Posters & Demonstrations Track*, vol. 1272, 2014, pp. 401–404.

[22] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange (HDT)," *J. Web Semant.*, vol. 19, pp. 22–41, 2013.

[23] L. Galárraga, K. Hose, and R. Schenkel, "Partout: a distributed engine for efficient RDF processing," in *WWW 2014*. ACM, 2014, pp. 267–268.

[24] L. Galárraga, K. A. Jakobsen, K. Hose, and T. B. Pedersen, "Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets," in *ISWC*, 2018, pp. 547–565.

[25] O. Görlitz and S. Staab, "SPLENDID: SPARQL endpoint federation exploiting VOID descriptions," in *(COLD2011)*, 2011.

[26] D. Graux, G. Sejdiu, H. Jabeen, J. Lehmann, D. Sui, D. Muhs, and J. Pfeffer, "Profiting from kitties on ethereum: Leveraging blockchain RDF with SANSA," in *ISWC Posters and Demos*, 2018.

[27] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in SPARQL queries," in *EDBT 2014*, 2014, pp. 439–450.

[28] E. R. Hansen, M. Lissandrini, A. Ghose, S. Løkke, C. Thomsen, and K. Hose, "Transparent Integration and Sharing of Life Cycle Sustainability Data with Provenance," in *ISWC*, vol. 12507, 2020, pp. 378–394.

[29] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich, "Data summaries for on-demand queries over linked data," in *WWW 2010*. ACM, 2010, pp. 411–420.

[30] O. Hartig and C. Buil-Aranda, "Bindings-restricted triple pattern fragments," in *OTM Conferences*, 2016.

[31] O. Hartig and et al., "RDF-star and SPARQL-star. W3C Draft Community Group. Report. W3C Community," 2021. [Online]. Available: https://w3c.github.io/rdf-star/cg-spec/2021-12-17.html

[32] L. Heling and M. Acosta, "A framework for federated SPARQL query processing over heterogeneous linked data fragments," *CoRR*, vol. abs/2102.03269, 2021. [Online]. Available: https://arxiv.org/abs/2102.03269

[33] L. Heling, M. Acosta, M. Maleshkova, and Y. Sure-Vetter, "Querying large knowledge graphs over triple pattern fragments: An empirical study," in *ISWC 2018*, 2018, pp. 86–102.

[34] D. Hernández, L. Galárraga, and K. Hose, "Computing How-Provenance for SPARQL Queries via Query Rewriting," *Proc. VLDB Endow.*, vol. 14, no. 13, pp. 3389–3401, 2021.

[35] J. V. Herwegen, R. Verborgh, E. Mannens, and R. V. de Walle, "Query execution optimization for clients of triple pattern fragments," in *ESWC 2015*, 2015, pp. 302–318.

[36] K. Hose, "Knowledge graph (r) evolution and the web of data," in *MEPDaW 2021*, 2021.

[37] K. Hose and R. Schenkel, "Towards benefit-based RDF source selection for SPARQL queries," in *SWIM 2012*. ACM, 2012, p. 2.

[38] ——, "WARP: workload-aware replication and partitioning for RDF," in *ICDE 2013 Workshops*, 2013, pp. 1–6.

[39] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi, "Processing Aggregate Queries in a Federation of SPARQL Endpoints," in *ESWC*, 2015, pp. 269–285.

[40] ——, "Optimizing Aggregate SPARQL Queries Using Materialized RDF Views," in *ISWC*, 2016, pp. 341–359.

[41] A. L. Jakobsen, G. Montoya, and K. Hose, "How diverse are federated query execution plans really?" in *ESWC 2019*, 2019, pp. 105–110.

[42] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi, "Atlas: Storing, updating and querying RDF(S) data on top of dhts," *J. Web Semant.*, vol. 8, no. 4, pp. 271–277, 2010.

[43] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John, "UniStore: Querying a DHT-based universal storage," in *ICDE 2007*, 2007, pp. 1503–1504.

[44] P. Larson, "Dynamic hash tables," *Commun. ACM*, vol. 31, no. 4, pp. 446–457, 1988.

[45] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee, "A demonstration of the solid platform for social web applications," in *WWW 2016*. ACM, 2016, pp. 223–226.

[46] T. Minier, H. Skaf-Molli, and P. Molli, "Sage: Web preemption for public SPARQL query services," in *WWW 2019*. ACM, 2019, pp. 1268–1278.

[47] G. Montoya, C. Aebeloe, and K. Hose, "Towards efficient query processing over heterogeneous RDF interfaces," in *DeSemWeb@ISWC 2018*, 2018.

[48] G. Montoya, I. Keles, and K. Hose, "Analysis of the effect of query shapes on performance over LDF interfaces," in *QuWeDa@ISWC 2019*, 2019, pp. 51–66.

[49] ——, "Querying linked data: An experimental evaluation of state-of-the-art interfaces," *CoRR*, vol. abs/1912.08010, 2019. [Online]. Available: http://arxiv.org/abs/1912.08010

[50] G. Montoya, H. Skaf-Molli, and K. Hose, "The odyssey approach for optimizing federated SPARQL queries," in *ISWC 2017*, pp. 471–489. [Online]. Available: https://doi.org/10.1007/978-3-319-68288-4_28

[51] G. Montoya, M. Vidal, and M. Acosta, "A heuristic-based approach for planning federated SPARQL queries," in *COLD 2012*, 2012.

[52] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[53] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *ICDE 2011*, 2011, pp. 984–994.

[54] O. Papapetrou, W. Siberski, and W. Nejdl, "Cardinality estimation and dynamic length adaptation for bloom filters," *Distributed Parallel Databases*, vol. 28, no. 2-3, pp. 119–156, 2010.

[55] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W. Han, "G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching," in *SIGMOD 2020*. ACM, 2020, pp. 1099–1114.

[56] J. Pérez, M. Arenas, and C. Gutiérrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 16:1–16:45, 2009.

[57] K. Rabbani, M. Lissandrini, and K. Hose, "Optimizing SPARQL Queries using Shape Statistics," in *EDBT 2021*, 2021, pp. 505–510.

[58] ——, "SHACL and ShEx in the Wild: A Community Survey on Validating Shapes Generation and Adoption," in *WWW'22 Companion, April 25–29, 2022, Virtual Event, Lyon, France*, 2022.

[59] M. Saleem, A. Hasnain, and A. N. Ngomo, "Largerdfbench: A billion triples benchmark for SPARQL endpoint federation," vol. 48, 2018, pp. 85–125.

[60] M. Saleem, A. Potocki, T. Soru, O. Hartig, and A. N. Ngomo, "Costfed: Cost-based query optimization for SPARQL endpoint federation," in *SEMANTiCS 2018*, 2018, pp. 163–174.

[61] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A. N. Ngomo, "How representative is a SPARQL benchmark? an analysis of RDF triplestore benchmarks," in *WWW 2019*. ACM, 2019, pp. 1623–1633.

[62] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "Fedx: Optimization techniques for federated query processing on linked data," in *ISWC 2011*, 2011, pp. 601–616.

[63] M. Sopek, P. Gradzki, W. Kosowski, D. Kuzinski, R. Trójczak, and R. Trypuz, "Graphchain: A distributed database with explicit semantics and chained RDF graphs," in *WWW Companion*, 2018, pp. 1171–1178.

[64] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over linked data," *World Wide Web*, vol. 14, no. 5-6, pp. 495–544, 2011.

[65] P. Vandenbussche, J. Umbrich, L. Matteis, A. Hogan, and C. B. Aranda, "SPARQLES: monitoring public SPARQL endpoints," *Semantic Web*, vol. 8, no. 6, pp. 1049–1065, 2017.

[66] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert, "Triple Pattern Fragments: A low-cost knowledge graph interface for the Web," *J. Web Sem.*, vol. 37-38, pp. 184–206, 2016.

[67] M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres, "Efficiently joining group patterns in SPARQL queries," in *ESWC 2010*, 2010, pp. 228–242.

[68] D. Vrandecic and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Commun. ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[69] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: a survey," *IJWGS*, vol. 14, no. 4, pp. 352–375, 2018.

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-863-2

AALBORG UNIVERSITY PRESS