Exploration of Large-Scale SPARQL Query Collections: Finding Structure and Regularity for Optimizing Database Systems

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Thomas Timm

aus Herne

1. Gutachter: Prof. Dr. Wim Martens
2. Gutachter: Prof. Dr. George Fletcher

Tag der Einreichung: 20. August 2019
Tag des Kolloquiums: 19. Februar 2020

Erforschung von riesigen SPARQL Anfragesammlungen:
Erkennung von Struktur und Regularität zur
Optimierung von Datenbanksystemen

Deutsche Kurzfassung

Nachdem das Word Wide Web erfolgreich in das Leben von Menschen überall Einzug gehalten hatte, bereitete es den Weg für das Semantic Web. Während das Word Wide Web zur menschlichen Nutzung konzipiert worden inst, soll das Semantic Web Maschinen die Verarbeitung von Daten erleichtern. Zu diesem Zweck werden Daten als Ontologie statt mit klassischen relationalen Datenbanken modelliert.

Die vorliegende Arbeit befasst sich mit der Erforschung von riesigen Sammlungen von Anfragen für semantische Datenbanken. Konkret wurden *über eine halbe Milliarde Anfragen* untersucht. Die vom *World Wide Web Consortium* (W3C) entwickelte Spezifikation *Resource Description Framework* (RDF) hat sich als Standard zur Abbildung von semantischen Daten etabliert. Als zugehörige Sprache für Anfragen ist das *SPARQL Protocol and RDF Query Language* (SPARQL, ein rekursives Akronym) vom W3C entwickelt worden.

Es gibt mehrere große öffentliche Datenbanken die semantische Daten zur Verfügung stellen. Die Nutzung dieser Endpunkte ergibt sich aus deren Protokollierung. Diese Aufzeichnungen liefern Anfragesammlungen, die Aufschluss über die tatsächliche Nutzung der Daten und der Funktionen von SPARQL geben können. Untersucht wurden zwei primäre Quellen für Anfragen: Eine gemischte Sammlung, die hauptsächlich von USEWOD stammt, und öffentlich verfügbare Anfragesammlungen von Wikidata. Die gemischte Sammlung besteht hauptsächlich aus Anfragen von DBpedia, aber sie enthält auch Quellen wie LinkedGeoData, OpenBioMed und BioPortal.

Das Ziel der Untersuchungen in dieser Arbeit ist die Daten der Anfragesammlungen zu ordnen und deren Inhalt zu verstehen, so dass Trends erkannt und Einsichten aus den Anfragesammlungen gewonnen werden können, welche Verwendung haben, um auch zukünftige Richtungen zur Erforschung und Optimierung von Datenbanken und verwandten Technologien aufzuzeigen. Leitende Fragen bei der Forschung kommen aus dem Gebiet der Evaluierung, Optimierung und Leistungsmessung von Anfragen.

Es stellt sich heraus, dass mehrere Beobachtungen getroffen werden

können und daraus verschiedene interessante Schlüsse gezogen werden können. So ist etwa eine sehr große Anzahl von Anfragen sehr einfach. Ferner ist es möglich, die Form der meisten auch komplexeren Anfragen durch Formen zu beschreiben, die günstige Eigenschaften hinsichtlich der Evaluierung haben. Ferner ist es möglich, Unterschiede bei Anfragen zu sehen, je nachdem ob sie von Menschen oder Maschinen stammen.

Es gab in der Vergangenheit bereits einige Studien von SPARQL Sammlungen [Ari+11]; [Han+16]; [Möl+10]; [Sal+15], aber der Fokus lag dabei lediglich auf statistischen Eigenschaften. Wikidata Anfragen wurden kürzlich als erstes von Malyshev et al. [Mal+18] und Bielefeld et al. [BGK18] untersucht. Allerdings gibt es mehrere Unterschiede bei diesen Untersuchungen. Sie führen keine tiefe strukturelle Untersuchung wie in dieser Arbeit durch.

In dieser Arbeit werden mehrere neuartige Ansätze und Techniken vorgestellt, wie die Analyse der Formen von Anfragen, der Studie von temporalen Aspekten in Anfragesammlungen oder die Untersuchung der Ähnlichkeit in Strukturen von Anfragen. Die Ergebnisse dieser Arbeit sind vollständig reproduzierbar, die entsprechende Software ist unter einer Open-Source Lizenz veröffentlicht und bietet weitere Funktionen zur Erkundung und Erforschung von Anfragesammlungen.

Wir werden nun eine Übersicht zu den Kapiteln geben.

Kapitel 2 gibt eine Übersicht über die vorhandenen Daten und deren Eigenschaften durch Analysen unter einfacheren Gesichtspunkten. Dazu führt es zunächst die verschiedenen Endpunkte, d.h. die Quellen, für die Daten ein. Danach werden Grundlagen für die späteren formalen Schritte erklärt. Damit ausgestattet werden Analysen zu Größen und Zusammensetzung der Bestandteile und Funktionen der Anfragen ausgeführt.

Die prominentesten Quellen für Anfragen in dieser Arbeit sind Wikidata und von USEWOD zur Verfügung gestellte Datensätze, darunter besonders von DBpedia, aber auch andere Quellen wie LinkedGeoData, OpenBioMed und BioPortal (Abschnitt 2.1). Bei den Analysen werden dabei als Einteilung zwei große Datensätze unterschieden: Zum einen die Anfragen von heterogenen Quellen von vornehmlich USEWOD, zum anderen die homogenen Anfragen von Wikidata.

Bevor eine komplexere strukturelle Analyse vorgenommen wird, beschäftigt sich Kapitel 3 gesondert mit den Prädikaten in den Anfragen, ein Aspekt, der für die spätere strukturelle Analyse wichtig ist, besonders für Anfragen von Wikidata. Prädikate in SPARQL können nicht nur aus einfachen Atomen

bestehen, sie können eine Form von regulären Ausdrücken annehmen, was Auswirkungen auf die Auswertung von Anfragen haben kann, so dass diese Frage von Bedeutung ist.

Nach der Betrachtung von allgemeinen Eigenschaften von Anfragen wird in **Kapitel 4** schließlich auf die inneren Strukturen und die damit verbundenen Komplexität eingegangen. Dabei werden zunächst die Ergebnisse aus der Vorarbeit bei der allgemeinen Betrachtung relevant, es wird erläutert wie die Zusammensetzung der Anfragen eine differenzierte Einteilung erfordert. Jeder Operator in SPARQL muss berücksichtigt werden, da eine bestimmte Verwendung dazu führen kann, dass sich die Komplexität der Anfrage ändert. So wurde insbesondere zuvor erforscht, wie die Operatoren für optionale Elemente und zur Filterung Auswirkungen haben können.

Nach der Formulierung eines Plans zur Einteilung der Anfragen wird die Struktur von Anfragen schließlich untersucht. Dabei soll die Struktur Aufschluss über die Komplexität der Anfragen liefern. Viele Anfragen lassen sich dafür als Graph betrachten, es gibt allerdings Ausnahmen, die eine erweiterte Betrachtung als Hypergraph erfordern. Es stellt sich heraus, dass eine große Anzahl von Anfragen sehr einfach ist. Sie bestehen oft aus nur einem Tripel, sie lassen sich sehr oft mit einer Baumstruktur beschreiben. Wenn sie zyklisch sind, dann oft nur sehr eingeschränkt.

Ferner stellt sich heraus, dass die meisten Anfragen sich mit einer Struktur beschreiben lassen, die aufgrund ihres Aussehens als „Blume“ bezeichnet wird. Als Maß für die Komplexität von zyklischen Anfragen wird die sogenannte Baumweite für Graphen und Hypergraphen (Abschnitt 4.2.2) herangezogen, was die Aussagen genauer quantifiziert. Für andere Strukturen werden ebenfalls andere Maße wie Tiefen und Verzweigungen (Abschnitt 4.2.3) zur genaueren Quantifizierung untersucht.

In **Kapitel 5** wird eine Reihe gänzlich neuer Ansätze vorgestellt. Dabei wechselt die Sicht der Analyse. Anstatt einzelne Anfragen zu untersuchen wird versucht, die Anfragen in der Gesamtheit zu analysieren. Die Fragestellung ist, ob es in den Anfragesammlungen einen temporalen Zusammenhang von Anfragen gibt. Dabei werden Erkenntnisse über die Entwicklung von Anfragen gewonnen, es werden Aspekte wie Fehler und Veränderungen der Form und Größe betrachtet.

Für diese Untersuchung ist ein Maß für die Ähnlichkeit von Anfragen wichtig. Dabei wird zunächst ein Zeichenketten-basierter Ansatz genommen,

danach wird ein Ansatz auf Basis des Syntaxbaums einer Anfrage vorgestellt, welcher effizient für die Suche von Anfragen ist.

Kapitel 6 befasst sich mit der Software, die für die Erforschung aller vorangegangen Fragen entwickelt wurde. Die Software ist unter einer Open-Source Lizenz veröffentlicht und kann somit kontrolliert, verwendet und weiterentwickelt werden. Unter Verwendung von öffentlichen Datensätzen wie von Wikidata lassen sich sämtliche Ergebnisse reproduzieren. Das Kapitel gibt einen Überblick über die Architektur und die Reproduzierbarkeit der Ergebnisse dieser Arbeit. Für die Erforschung der Anfragen war es zunächst wichtig, überhaupt Regelmäßigkeiten zu erkennen, so dass die Software Werkzeuge zum Auffinden von Erkenntnissen entwickelt wurde. Dieser interaktive Software-Aspekt wird ebenfalls vorgestellt.

Zusammenfassend wird in **Kapitel 7** noch einmal auf die Ergebnisse und Erkenntnisse zurück geblickt und zukünftige Forschungsaspekte angesprochen.

Die meisten Ergebnisse, die auf dem heterogenen Datensatz basieren, sind in [BMT17a] veröffentlicht worden. Darauf aufbauend wurde eine signifikante Erweiterung dieser Untersuchungen in [BMT19a] veröffentlicht. Die meisten Ergebnisse zu Wikidata Datensätzen wurden in [BMT19b] veröffentlicht. Der verwendete Code und zusätzliche Erweiterungen wurden mit [BMT18] veröffentlicht. Die Veröffentlichungen sind aus einer Zusammenarbeit mit Angela Bonifati und Wim Martens entstanden. Die gesamten technischen Umsetzungen sind dabei ausschließlich vom Autor dieser Arbeit durchgeführt worden.

Da in den Publikationen nicht nur andere Datensätze betrachtet wurden, sondern auch unterschiedliche Aspekte untersucht wurden, sind in dieser Arbeit die Aspekte ergänzt worden, die in anderen Publikationen nicht vorhanden waren, oder die aus Platzgründen in den Veröffentlichungen ausgelassen wurden. Zusätzlich wurden weitere Aspekte zu dem Aufbau der Software ergänzt.

Acknowledgements

First, I would like to thank my advisor Wim Martens. He has an incredible attention to detail, which greatly helped to drive me to always strive for most comprehensive results. He provided great insights with his vast knowledge, which inspired me to look at many different sides of a problem, and he was always readily available to give advice on any and all issues, which supported me greatly in my work.

Next, I would also like to thank Angela Bonifati for her support. She always provided valuable insight and ideas for directions regarding research questions. This greatly shaped my research.

Personally, I want to especially thank my parents for their support. Finally, I would like to thank the rest of my family and my friends for always supporting me in my endeavor.

Herne, August 2019

Contents

1

Introduction

After the World Wide Web successfully penetrated the lives of people everywhere, it gave rise to the Semantic Web. Whereas the World Wide Web started to be used by humans, the Semantic Web is meant to facilitate machines to process data. To this end, data is modelled as ontology as opposed to storing it in classical relational databases.

The work presented here deals with the research of large-scale collections of queries for semantic databases. Specifically, *more than half a billion queries* are investigated. The *World Wide Web Consortium* (W3C) specification *Resource Description Framework* (RDF) became the prominent standard for modelling semantic data. As corresponding language for querying, the *SPARQL Protocol and RDF Query Language* (SPARQL, a recursive acronym) was developed by the W3C.

There are various large-scale public databases that offer semantic data for querying. These public endpoints log their usage for various purposes. These logs can offer insight into the actual usage of data and features in SPARQL. We investigate two primary sources for queries: A diverse collections mostly obtained from USEWOD, and publicly available query logs from Wikidata. The diverse collections consists mostly of logs from DBpedia, but it also includes sources such as LinkedGeoData, OpenBioMed, and BioPortal.

The goal of the study in this work is to organize the data in the logs to make sense of it, so trends and insights on the nature of queries in the logs can be identified, which can be used to derive future directions for optimizing database systems that handle linked data and technology surrounding this topic. Therefore, questions guiding the research are from topics such as query evaluation, query optimization, tuning, and benchmarking.

It turns out that quite a few observations can be made and it allows to draw several interesting conclusion. For instance, a very large number of queries is extremely simple. It is possible to describe the shapes of most

queries, even more complex ones, with a shape that has favorable properties regarding the efficiency of evaluation. Furthermore, there are differences in queries originating from humans when compared to machine-generated queries.

There have been studies of SPARQL query logs in the past, such as [Ari+11]; [Han+16]; [Möl+10]; [PV11]; [Sal+15], but the focus was mainly focused on statistical features. Wikidata logs have been studied first rather recently by Malyshev et al. [Mal+18] and Bielefeld et al. [BGK18]. However, there are several differences in their study. They do not perform a deep structural study that is performed in this work.

In this work, several novel, new approaches are taken such as the analysis of shapes of queries, the study of logs with a temporal analysis, and the investigation of query similarity based on structure. Results are entirely reproducible, the accompanying software is made available under an open-source license, and it can be used to explore logs in addition to analyzing them.

We will now give a short overview of the chapters.

Chapter 2 gives and overview of the data sets at hand and their properties by looking at more general properties. To this end, first the different endpoints are introduced, i.e. the sources for queries. Next, basic concepts for further steps are introduced. With this, we run through a series of analyses for insights on size and composition of various parts and features found in queries.

The two primary sources for queries in this work are Wikidata and data sets provided by USEWOD, and for it most notably DBpedia, but it also includes sources such as LinkedGeoData, OpenBioMed, and BioPortal (Section 2.1). For the analyses, two collections of data sets are considered as main division: on one hand the queries from heterogeneous sources mainly from USEWOD, on the other hand the homogeneous queries from Wikidata.

Before turning to a more complex, structural analysis, **Chapter 3** deals with predicates in SPARQL separately, which will become important in the structural analysis later, especially for queries in Wikidata. Predicates in SPARQL can not only consist of simple atoms, they can also use a form of regular expressions, which can have an impact on the evaluation of queries, which explains the importance of exploring this topic.

After examining general properties of queries, **Chapter 4** finally turns to the inner structure of queries and the complexity resulting from it. To this end, the groundwork from previous results we examined in our more general results become relevant again, we start looking at the composition of queries in order to get a sensible partitioning of our data. Operators in SPARQL must be carefully considered, because certain usages can lead to changes in complexity for evaluating queries. For instance, previously, the study of operators for optional elements and filtering were a topic of intensive research.

After formulating a plan for dividing our queries, the inner structure of queries is studied. More specifically, the structure should provide insight into the complexity of queries. Most queries can be interpreted as a regular graph, but there are exceptions that require an extended study of queries as hypergraphs. It turns out that an overwhelming majority of queries is extremely simple. Often they can consist of only a single triple, their shape can often be captured with trees. If they have any cyclicity, they are often only mildly cyclic.

Additionally, it turns out that most queries can be captured with a simple shape that is called "flower", named after its appearance. A measure for complexity of cyclic queries is treewidth in the context of graphs and hypergraphs (Section 4.2.2), which can be used to quantify this more precisely. Other measures for queries such as depth and branching (Section 4.2.3) are studied as well.

Chapter 5 introduces several completely new novel approaches. We switch from considering only isolated queries with no regard for the order in logs to investigating query logs as a whole. The goal of this is to find out if a temporal aspect in queries can be observed. New insights can be gained from this, we will examine aspects such as errors and changes in form and sizes of queries. For this research, a measure for query similarity is important. We begin with a string-based approach, then we turn to an approach based on the abstract syntax tree of queries, which is also useful for searching for queries efficiently.

Chapter 6 focuses on the software that was developed for studying all previous research questions. The code for the software is made available as open-source, and as such, results can be confirmed, and it can be used and extended. By taking public data sets such as Wikidata, the results presented in this work are fully reproducible. The chapter gives an overview about the

architecture of the software and outlines how to reproduce the results for validation. When starting out the research of queries, it was important to find any kind of regularity and trends with the queries, which lead to the development of several tools to aid in this. This interactive software aspect is presented as well.

Concluding things, in **Chapter 7** the results and insights that are gained from them are reflected upon, and future work is discussed.

Most results for the heterogeneous data sets are published in [BMT17a]. Building on these results, a significant extension of these studies is published in [BMT19a]. Most results dealing with Wikidata queries are published in [BMT19b]. The code for obtaining results, along with additional extensions, is published with [BMT18]. These publications resulted from a collaboration with Angela Bonifati and Wim Martens. The entire technical realization is done solely by the author of this work.

Because the publications not only deal with different data sets, but also with different aspects, this work contains additional results to complete these gaps, which are not part of the published results, either because they are new or were omitted for space reasons. Additionally, more details regarding the software have been added.

1.1 Motivation, Overview, and Background

Ontological Query Logs As more and more data is exposed in RDF format, we are witnessing a compelling need from end-users to formulate more or less sophisticated queries on top of this data. SPARQL endpoints are increasingly used to harvest query results from available RDF data repositories. But how do these end-user queries look like? As opposed to RDF data, which can be easily obtained under the form of dumps (DBpedia and Wikidata dumps [DBP17]; [VK14]; [Wik17]), query logs are often inaccessible, yet hidden treasures to understand the actual usage of these data. In this work, we investigate a large corpus of query logs from different SPARQL endpoints, which spans over several years (2009–2017). In comparison to previous studies on real SPARQL queries [Ari+11]; [Han+16]; [Möl+10]; [PV11]; [Sal+15], which typically investigated query logs of a single source, in this work, a multi-source query corpus that is two orders of magnitude larger is considered. Furthermore, the analysis goes significantly deeper. In particular, it is the first to do a large-scale analysis on the topology of queries, which has seen significant theoretical interest in the last decades (e.g., [CR97]; [GLS02];

[Got+16]) and is now being used for state-of-the-art structural decomposition methods for query optimization [Abe+16a]; [Abe+16b]; [KEK17]. As a consequence, this is the first analytical study on real (and most recent) SPARQL queries from a variety of domains reflecting the recent advances in theoretical and system-oriented studies of query evaluation.

The work makes the following contributions. Apart from classical measures of syntactic properties of the investigated queries, such as their keywords, their number of triples, and operator distributions, which is also applied to the corpus, we also mine the usage of projection in queries and subqueries in the various data sets. Projection indeed is the cause of increased complexity (from PTIME to NP-Complete) of the following central decision problem in query evaluation [BPS15]; [CM77]; [Let+13].

We then proceed by considering queries under their graph and hypergraph structures. Such structural aspects of queries have been investigated in the database theory community for over two decades [Got+16] since they can indicate when queries can be evaluated efficiently. Recently, several studies on new join algorithms leverage the hypergraph structure of queries in the contexts of relational and RDF query processing [Abe+16a]; [KEK17]. Theoretical research in this area traditionally focused on *conjunctive queries (CQs)*. For CQs, we know that tree-likeness of their structure leads to polynomial-time query evaluation [Got+16]. For larger classes of queries, the topology of the graph of a query is much less informative. For instance, if we additionally allow SPARQL's Optional operator, evaluation can be NP-complete even if the structure is a tree [BPS15]. For this reason, we focus our structural study on CQ-like queries. We develop a shape classifier for such queries and identify their most occurring shapes. Interestingly enough, these queries have quite regular shapes. The overwhelming majority of the queries is acyclic (i.e., tree- or forest-shaped). We discovered that the cyclic queries mostly consist of a central node with simple, small attachments (which we call *flower*). In terms of tree- and hypertreewidth, we discovered that the cyclic queries have width two, up to a few exceptions with width three.

In order to gauge the performances of cyclic and acyclic queries from a practical viewpoint, we have run a comparative analysis of chain and cycle queries synthetically generated with an available graph and query workload generator [Bag+17]. This experiment showed different behaviors of SPARQL query engines, such as Blazegraph and PostgreSQL with query workloads of CQs of increasing sizes (intended as number of conjuncts). It also lets us grasp a tangible difference between chain and cycle queries in either query engine, this difference being more pronounced for PostgreSQL. We may interpret this result as a lack of maturity of practical query engines for cyclic queries, thus motivating the need of specific query optimization techniques

for such queries as in [Abe+16a]; [KEK17].

Finally, we deal with the problem of identifying sequences of similar queries in the query logs. These queries are then classified as gradual modifications of a seed query, possibly by the same user. We measure the length of such streaks in three log files from DBpedia. We conclude our study with insights on the impact of our analytical study of large SPARQL query logs on query evaluation, query optimization, tuning, and benchmarking.

Wikidata Wikidata [VK14] is a free collaborative knowledge base that has been characterized by a gigantic growth in terms of number of edits, number of users and developers, and amount of automated software since its inception in 2012 by the Wikimedia Foundation. The interplay between user and bot activities on Wikidata is an interesting subject to study in order to make sense of the quality of the newly added items produced by the massive numbers of edits in the knowledge base [Pis18]. Contrarily to Wikidata data dumps, which are readily available and allow a battery of analyses, the activity of both humans and bots on the Wikidata SPARQL endpoints can only be investigated since recently, thanks to the release of large anonymized query logs.These query logs represent a rich set of information about the robotic and organic query traffic on Wikidata and deserve our attention for further investigation, in particular to understand the structure of complex queries.

A preliminary analysis of the Wikidata query logs bootstrapped with a recent paper by Malyshev et al. [Mal+18], who first introduced the Wikidata SPARQL service WDQSand pinpointed its technical characteristics and current usage. They also provided a classification of the Wikidata queries into robotic and organic requests that are readily adopted in the present work. They made several observations on which this work builds, namely that robotic query traffic dominates organic query traffic in terms of volume and query load, and that robotic queries are issued by a single source, whereas organic queries are typically of multi-source origin. They also identified a massive presence of recursive queries in these logs, with a prominent fragment that consists of queries only containing joins and property paths, also known as conjunctive 2-way regular path queries (C2RPQs) in literature. These massive logs of recursive queries are the first encountered so far. C2RPQs are the basic building blocks of graph query languages in the literature of RDF and graph databases. They allow to express navigational patterns on the graph instances by leveraging regular expressions, also known as Property Paths in the SPARQL 1.1 specification [HS13]. In Wikidata, they are particularly important since they emulate ontological reasoning in

SPARQL and also express complex label-constrained reachability queries.

The analysis in this work focuses on these recursive queries in the Wikidata logs and further extend the class of C2RPQs by incorporating the use of Service, Values, Bind, Filter, and Optional. Indeed, Service, Values, and Bind occurred only rarely in other massive logs [BMT17a], but are very prominent in Wikidata query logs. This fragment, which we call C2RPQ+, constitutes more than 85% of the valid queries in the logs.

Although the distinction between robotic and organic query traffic has been introduced in [Mal+18], it has never been used in a complex analysis such the one that is carried out in this work regarding the types of property paths, the computation of triples when property paths are present, and the shape analysis of the C2RPQ+ fragment. A view from rather different angles of these logs is presented by considering them with or without duplicates, and by separating the analysis of successfully executed and timeout queries, the latter being analyzed for the first time in our study. Additionally, to aid work, a query similarity search tool was developed capable of identifying from an initial query the set of structurally similar queries by using tree edit distance. This tool allows to further inspect the logs by having a specific query in mind, and in a sense permits to reproduce and reapply the previous complex analysis to the obtained sets for future studies, because it allows to pinpoint and extract queries from huge logs.

This study focuses on the following research questions: What is the distribution of query sizes? Which qualifiers are popular in queries? How are property paths used and what is their structure? How prominent are conjunctive queries (and variants thereof)? What is the shape and (hyper)treewidth of queries? Given a query, can we find in a subset of the logs the queries that are structurally similar to it? Furthermore, we are also interested in meta-questions, such as: Are there differences between robotic and organic queries?

All presented results are entirely reproducible. The code for the analysis is publicly available [Tim18].

1.2 Related Work

This work is mainly based on results published in a series of articles [BMT17a]; [BMT18]; [BMT19a]; [BMT19b]. However, there are additions to the results from the articles.

Studies on Query Logs Several studies of SPARQL queries in the past have mainly focused statistical features of the queries, such as occurrences

of triple patterns, types of queries, or query fragments [Ari+11]; [Han+16]; [Möl+10]; [Sal+15].

The only early study that investigated the relationship between structural features of practical queries and query evaluation complexity has been conducted by Picalausa and Vansummeren [PV11]. However, it focused on a smaller corpus (3M queries from DBpedia 2010), which make the findings less suitable to be generalized.

USEWOD and DBpedia data sets have also been considered in [Ari+11]. It takes into account the log files from DBpedia and SWDF reaching a total size of 3M. The work mainly investigates the number of triples and joins in the queries. Based on the observation of [NW10] that SPARQL graph patterns are typically chain- or star-shaped, they also look at their occurrences. They found very scarce chains and high coverage of almost star-shaped graph patterns, but they do not characterize the latter.

A query analysis and clustering of DBpedia SPARQL queries has been performed in [Mor+11] in order to build a set of prototypical benchmarking queries. Query logs have been inspected in a user study in [HP15] to understand whether facts that are queried together provide intra-fact relatedness in the Linked Open Data graph. The objectives of both papers are different from the one pursued in this work.

Finally, [Alj+14] investigates on the use of regular expressions in the Filter clause of USEWOD queries.

Wikidata Logs Recently, Malyshev et al. [Mal+18] and Bielefeld et al. [BGK18] were the first to analyze a massive collection of Wikidata queries. Malyshev et al. first introduced the Wikidata SPARQL service WDQS and analyzed basic characteristics of Wikidata queries related to their usage in this service spanning fromSPARQL feature prevalence and correlation to annotations and language distributions. They also isolated the robotic and organic queries, however, this was mostly derived from the metadata of the queries. This classification considers the queries issued by a browser as organic and the remaining queries as robotic. Additionally, some high-volume queries issued from the same source in a very short timeframe are considered as robotic. The classification that is done by them is not applicable for most DBpedia logs, because they lack the information about browser- and machine-generated traffic, or they have been modified for anonymization purposes.

They also identified the C2RPQs fragment, i.e. the largest fragment encountered so far of conjunctive 2-way regular path queries. In [BMT19b], the first large-scale structural analysis of this gigantic query collection is per-

formed. A structural classification of real-world property paths is done on the first large set of property paths relevant for Wikidata, and 57x larger than the set considered mainly for DBpedia in [BMT17a]. This also investigates the shape of C2RPQs, which would not be possible with the classification for only conjunctive queries (CQ) in [BMT17a]. The occurrences of property paths in the latter corpus is negligible with respect to the size of the C2RPQ fragment considered.

Theory The most important works for theory and concepts in this work will be discussed now.

Complexity issues caused by Optional are studied in several works. Especially important are the following ones. Kaminski and Kostylev [KK18] continue the investigation of well-designed queries and introduced the notion of weakly well-designed queries. Barceló et al. [BPS15] introduced the notion of well-designed pattern trees for queries and bounded interface width.

Gottlob investigated tree- and hypertree decomposition and the impact of tree- and hypertreewidth on query evaluation in several works [GGS14]; [GLS02].

Bagan and Bonifati [Bag+17]; [BBG13] investigate regular simple path queries on graphs and created gMark to create benchmarks for database systems, which is also used in this work to assess impact of query shapes.

1.3 Overview of Publications

This work covers material that has already been published. There is no direct mapping from publications to chapters, because the presentation is changed to present material interleaved side by side and new material has also been added. A listing of the publications, along with some notable contributions that can be attributed more uniquely to them are listed below:

1. [BMT17] contributes to Section 3.2 (property paths), and Section 5.1 (streaks).

2. [BMT18] contributes to Section 6.3 (DARQL).

3. [BMT19a] contributes to Section 3.2 (property paths), Section 5.1 (streaks), Section 4.2.3 (measuring shapes), and Section 3.4 (tree patterns).

4. [BMT19b] contributes to Section 3.1 (property paths), Section 5.2 (query search), and Section 3.3 (property paths).

5. [BMT19c] contributes to Section 6.4 (SHARQL).

6. [Tim18] item contributes to Section 6.3 (DARQL).
   It contains the code used for [BMT17] and [BMT18].

7. [Tim19] contributes to Section 6.4 (SHARQL).
   It contains the code used for [BMT19a] and [BMT19b]
   (12.8k source lines of code, 1.6k lines of comments).

The last two items are the published repositories of the source code, which has been made available under open-source licenses. All experiments, benchmarks, and the collection and preparation of analysis results were solely conducted by the author of this work.

List of Publications

[BMT17] A. Bonifati, W. Martens, and T. Timm. "An Analytical Study of Large SPARQL Query Logs". In: *PVLDB* 11.2 (2017), pp. 149–161. DOI: 10.14778/3167892.3167895.

[BMT18] A. Bonifati, W. Martens, and T. Timm. "DARQL: Deep Analysis of SPARQL Queries". In: *The Web Conference (WWW)*. 2018, pp. 187–190. DOI: 10.1145/3184558.3186975.

[BMT19a] A. Bonifati, W. Martens, and T. Timm. "An Analytical Study of Large SPARQL Query Logs". In: *The VLDB Journal* (2019). Extended version of [BMT17]. DOI: 10.1007/s00778-019-00558-9.

[BMT19b] A. Bonifati, W. Martens, and T. Timm. "Navigating the Maze of Wikidata Query Logs". In: *The Web Conference (WWW)* (2019). DOI: 10.1145/3308558.3313472.

[BMT19c] A. Bonifati, W. Martens, and T. Timm. "SHARQL: Shape Analysis of Recursive SPARQL Queries". Unpublished. 2019.

[Tim18] T. Timm. *DARQL*. https://github.com/PoDMR/darql/. 2018.

[Tim19] T. Timm. *SHARQL*. https://github.com/PoDMR/sharql/. 2019.

2

Basic Analysis

We will begin by making ourselves familiar with the data we have, and the domain surrounding it. The query logs we are dealing with are real-world ontological query logs. The following sections are meant to provide an overview before we dive into the compositions of queries.

2.1 Data Sets

There are two distinctive collections of data sets that were investigated separately. The first collection of data sets is a diverse collection of query logs consisting of data from various sources over various years. Most data sets were obtained from USEWOD and the largest amount of logs in them were from DBpedia. They also contained query logs from other sources. Some logs were augmented with more independently obtained logs for purposes such as additional timespans or spans. Additionally, a few other publicly available logs were also added to increase the corpus for investigation. This collection of data sets will be referred to as the *Multi-Source Collection*.

The second major collection of data sets is from Wikidata and will simply be called the *Wikidata Collection*. It makes sense to divide the analysis and discussion between these two collections, and indeed, we will see that these sets have some different characteristics. Both collections will be discussed in more detail in the following.

2.1.1 Multi-Source Collection

The Multi-Source Collection has a total of 350,089,005 queries, which were obtained as follows. The 2013–2016 USEWOD query logs, some additional DBpedia query logs for 2013, 2014, 2015, 2016, and 2017 were directly ob-

Source	*Total #Q*	*Valid #Q*	*Unique #Q*
DBpedia9-12	28,651,075	27,622,233	13,437,966
DBpedia13	5,243,853	4,819,837	2,628,000
DBpedia14	37,219,788	33,996,486	17,217,416
DBpedia15	43,478,986	42,709,781	13,253,798
DBpedia16	15,098,176	14,687,870	4,369,755
DBpedia17	169,110,041	164,297,723	34,440,636
LGD13	1,927,695	1,531,164	357,843
LGD14	1,999,961	1,951,973	628,640
BioP13	4,627,270	4,624,449	687,773
BioP14	26,438,932	26,404,716	2,191,151
BioMed13	883,375	882,847	27,030
SWDF13	13,853,604	13,670,550	1,229,759
BritM14	1,555,940	1,545,643	135,112
Wikidata17	309	308	308
Total	350,089,005	338,745,580	90,605,187

Table 2.1: Sizes of logs in Multi-Source Collection.

tained from Openlink[1], the 2014 British Museum query logs from LSQ[2], and user-submitted example queries from Wikidata[3] were extracted via crawling and scripting in February 2017. These log files are associated with 7 different data sources from various domains: DBpedia, Semantic Web Dog Food (SWDF), LinkedGeoData (LGD), BioPortal (BioP), OpenBioMed (BioMed), British Museum (BritM), and Wikidata.

Table 2.1 gives and overview of the analyzed query logs from the Multi-Source Collection, along with their main characteristics such as total number of queries, parsable number of queries, and the number of queries after deduplication. Since the logs for DBpedia were obtained from different sources, they were grouped as follows. `DBpedia9-12` contains the DBpedia logs from USEWOD'13, which are query logs from 2009–2012. All other `DBpedia'X` sets contain the query logs from the year `'X`, be it from USEWOD or from

[1] `http://www.openlinksw.com`

[2] `http://aksw.github.io/LSQ/`

[3] `https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`

Openlink.[4] Notice that for the data set for DBpedia from 2017 obtained from Openlink contributes 169,110,041 queries, which is a very significant amount of queries compared to the rest of the corpus. This number is reflected in the `DBpedia17` data set.

Preparation The logs were prepared for the analysis as follows. First the logs were cleaned by removing entries that are not queries (e.g., http requests). In the following only the actual SPARQL queries in the logs are reported. For each of the logs, the table summarizes the total number of queries (*Total*) and the number of queries that we could parse using Apache Jena 3.7.0 (*Valid*). From the latter set, duplicate queries were removed after whitespace normalization, resulting in the unique queries that could be parsed (*Unique*). The results on both *Valid* and *Unique* data sets will be presented. In summary, the corpus of query logs contains the latest blend of USEWOD and Openlink DBpedia query logs (the latter providing 51M more queries in the period 2013–2016 than the USEWOD corpus, and 169M more for 2017), plus BritM and Wikidata queries. Finally, although the online Wikidata example queries (Feb 13th, 2017) are a manually curated set, there was one query that could not be parsed.[5]

Notation For the Multi-Source Collection, the following notation will be used to discuss results on the *Valid* and *Unique* data sets. When a number or a percentage is reported in the format X (Y), the number X refers to the *Valid* and the number Y to the *Unique* set of queries. This notation allows the reader to stay informed about the queries that the endpoint actually receives (*Valid*) and about those without duplicates in this set (*Unique*).

Anonymization The obtained query logs are anonymized in the sense that they do not contain IP addresses, precise time stamps, or user agents. Time stamps are typically either completely absent, or rounded to an hour. For example in some of the logs, all time stamps are set to 3:00 GMT. This means, in particular, that these logs do not allow a classification into *robotic* and *organic* queries, as was done by Bielefeldt et al. [BGK18] and Malyshev et al. [Mal+18].

[4]Three log files were obtained both from USEWOD as well as from Openlink, as it turned out that only the hash values used for anonymization were different. These duplicate log files were deleted prior to all analysis and are not taken into account in Table 2.1.

[5]The query was called "Public Art in Paris" and was malformed (closing braces were missing and it had a bad aggregate). It was still malformed on June 29th, 2017.

Observations on Multi-Source Collection In the total Multi-Source Collection, 16,639,701 (2,978,945) queries, or 4.91% (3.29%) of the logs do not have a body. All these queries are Describe queries, and almost exclusively occur in `DBpedia14`–`DBpedia17`. To be more precise, 99.47% (97.22%) of the Describe queries do not have a body. Therefore some of the analyses are only conducted on Select, Ask, and Construct queries.

2.1.2 Wikidata Collection

The corpus of the Wikidata Collection consists of all queries in the Wikidata query logs that were recently made publicly available [Bie+18]. Precisely, the queries considered have been downloaded on October 12th, 2018. These logs are anonymized and represent queries that were submitted to the Wikidata SPARQL endpoint from June 12th until September 3rd in 2017. The same queries have been considered in the work of Malyshev et al. [Mal+18]. For analysis purposes, these log files were partitioned into four disjoint sets: Queries for which the HTTP request was successful (HTTP code 200) were split into organic (OrganicOK) and robotic queries (RoboticOK); timeout queries were, again, split into organic (OrganicTO) and robotic queries (RoboticTO).

Partitioning Each query in the downloadable log files has an annotation that indicates if it was classified as a bot or user query by Malyshev et al. [Mal+18]. This classification is used in the partitioning. The presented number of queries in Organic is slightly higher than the number of queries reported on the download page of the query logs. It is likely that that the Dresden file may be incomplete, since organic queries in the "all queries" log files can be found that do not show up in the organic subset in [Bie+18]. Sometimes OK, (resp., TO) is used to refer to OrganicOK $\cup$ RoboticOK (resp., OrganicTO $\cup$ RoboticTO) for brevity. The TO queries have not been considered in the work of Malyshev et al. [Mal+18].

Table 2.2 for the Wikidata Collection describes, for each of the log types, its number of queries (Total #Q), number of valid queries, i.e., queries that can be parsed using Apache Jena 3.7.0 (Valid #Q), and the number of valid queries after removal of duplicates (Unique #Q). For duplicate removal, two queries are considered to be the same if they are the same string after whitespace normalization and query prefix normalization (i.e. IRIs resolve to the same entities).

Like with the Multi-Source Collection, the same notation as discussed before in Section 2.1.1 is used to present *Valid* and *Unique* numbers.

Source	*Total #Q*	*Valid #Q*	*Unique #Q*
RoboticOK	207,505,296	207,464,954	34,523,883
OrganicOK	661,769	651,385	252,015
RoboticTO	33,616	33,465	3,168
OrganicTO	14,528	14,087	8,729
Robotic	207,538,912	207,498,419	34,527,051
Organic	676,297	665,472	260,744
OK	208,167,065	208,116,339	34,775,898
TO	48,144	47,552	11,897
Total	208,215,209	208,163,891	34,787,795

Table 2.2: Sizes of logs in Wikidata Collection.

Observations on Wikidata Collection From Table 2.2 a number of interesting observations can be made. One simple observation is that the robotic logs contain many more duplicates than the organic logs. Indeed, whereas Organic contains 38.69% unique queries, Robotic only contains 38.69% unique queries. A second observation is that, even though queries do not time out very often, organic queries time out 100 times more often than robotic queries. The fraction of OrganicTO queries to Organic queries is 2.12% (3.35%), whereas the fraction of RoboticTO to Robotic queries is 0.02% (0.01%). In the set of *unique* timeout queries, a staggering 73.37% are organic.

2.2 Preliminaries

Some basic definitions on RDF and SPARQL follow.

RDF RDF data consists of a set of triples $\langle s, p, o \rangle$ where s is referred to as *subject*, p as *predicate*, and o as *object*. According to the specification, s, p, and o can come from pairwise disjoint sets $\mathcal{I}$ (*IRIs*), $\mathcal{B}$ (*blank nodes*), and $\mathcal{L}$ (*literals*) as follows: $s \in \mathcal{I} \cup \mathcal{B}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$.

The precise definitions of IRIs, blank nodes, and literals are not important for our analysis. The most important thing to know is that blank nodes similar to *variables*, which is discussed later, while literals are constants like strings and numbers, and an IRI (Internationalized Resource Identifiers) is just an extension of an URI (Uniform Resource Identifier) supporting more characters (encoded UTF-8 instead of ASCII). So IRIs are just special strings to identify things.

SPARQL For our purposes, a *SPARQL query* Q can be seen as a tuple of the form

$$(\textit{query type}, \textit{pattern } P, \textit{solution modifier}).$$

Conceptually such queries work as follows: The central component is the *Pattern* P, which contains patterns that are matched onto the RDF data. The result of this part of the query is a multiset of mappings that match the pattern to the data.

The *solution modifier* allows aggregation, grouping, sorting, duplicate removal, and returning only a specific window (e.g., the first ten) of the multiset of mappings returned by the pattern. The result is a list L of mappings.

The *query type* determines the output of the query. It is one of four types: Select, Ask, Construct, and Describe. Select queries return projections of mappings from L. Ask queries return a Boolean, the answer is true *iff* the pattern P could be matched. Construct queries construct a new set of RDF triples based on the mappings in L. Finally, Describe queries return a set of RDF triples that describes the IRIs and the blank nodes in L. The exact output of Describe queries is implementation-dependent. Such queries are meant to help users explore the data. Compared to [PV11], we wil allow more solution modifiers and more complex patterns, as explained next.

Patterns Let $\mathcal{V} = \{?x, ?y, ?z, ?x_1, \ldots\}$ be an infinite set of variables, disjoint from $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$. As in SPARQL, variables will always be presented

prefixed by a question mark. A *triple pattern* is an element of $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{V}) \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$. A *property path* is a regular expression over the alphabet $\mathcal{I}$. A *property path pattern* is an element of $(\mathcal{I} \cup \mathcal{B} \cup \mathcal{V}) \times pp \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$, where pp is a property path. A *SPARQL pattern* is an expression generated from the following grammar:

$$\begin{aligned} P ::= \quad & t \mid pp \mid Q \mid P_1 \ \mathsf{And}\ P_2 \mid P \ \mathsf{Filter}\ R \\ & \mid P_1 \ \mathsf{Union}\ P_2 \mid P_1 \ \mathsf{Optional}\ P_2 \\ & \mid \mathsf{Graph}\ iv\ P \mid \mathsf{Values}\ tup\ T \end{aligned}$$

Here, t is a triple pattern, pp is a property path pattern, Q is again a SPARQL query, R is a so-called *SPARQL filter constraint*, and $iv \in \mathcal{I} \cup \mathcal{V}$. Property paths ($pp$) and subqueries ($Q$) in the above grammar are new features since SPARQL 1.1. SPARQL filter constraints R are built-in conditions which can have unary predicates, (in)equalities between variables, and Boolean combinations thereof. They are called filter expressions. The keyword Values binds a tuple *tup* to values in a given table T. Refer to the SPARQL 1.1 recommendation [HS13] and the literature [PAG09]for the precise syntax of filter constraints and the semantics of SPARQL queries. $\mathsf{vars}(P)$ will be used to denote the set of variables occurring in P.

Example query The following example illustrates how these definitions corresponds to real SPARQL queries. The following query comes from Wikidata ("Locations of archaeological sites", from [Wik17]).

```
SELECT ?label ?coord ?subj WHERE {
  ?subj wdt:P31/wdt:P279* wd:Q839954 .
  ?subj wdt:P625          ?coord .
  ?subj rdfs:label        ?label
  FILTER(lang(?label) = "en")
}
```

Listing 2.1: SPARQL query

The query uses the property path `wdt:P31/wdt:P279*`, literal `wd:Q839954`, and triple pattern `?subj wdt:P625 ?coord`. It also uses a filter constraint. In SPARQL, the And operator is denoted by a dot (and is sometimes implicit in alternative, even more succinct syntax). This is because in SPARQL, sets of triple patterns are conjunctive sets and the dot is just a separator.

Property Paths Property paths in SPARQL are very similar to regular expressions. Instead of symbols, property paths use IRIs as atoms. Like

regular expressions, they can be combined with concatenation (sequence, via a_1/a_2), choice (alternative, via $a_1|a_2$), or with repetition (zero/one or more, via $a*/a+$). They also feature atomic negation (via $!a$). Finally, they have a reverse navigation operator in the form of $\hat{}a$, which means "follow an a-edge in reverse direction".

Conjunctive Queries Conjunctive queries are a central class of queries in database research and on this class will be built on in subsequent sections. In the context of SPARQL, we define them as follows.

Definition 2.2.1. A *conjunctive query (CQ)* is a SPARQL pattern that only uses triple patterns and the operator And.

Another important query class is the following.

Definition 2.2.2. A *conjunctive 2-way regular path query (C2RPQ)* is a SPARQL pattern that only uses triple patterns, the operator And, and property paths.

Graphs A basic and very important structure that will be used in analyses are graphs.

Definition 2.2.3. A *graph* G is a pair (V, E) where V is a set of element called *vertices*, while E is a set of elements called *edges*. G is a *directed graph* if E is a set of pairs of vertices from V. G is called an *undirected graph* if E is set of a multisets of vertices from V with cardinality 2. A graph that uses multisets instead of sets for edges is called a *multigraph*, otherwise it is called a *simple graph*. An edge that appears more than once (by identity) in a multigraph is called a *parallel edge*. An edge with two identical vertices is called a *self-loop*.

The concept of graphs can be extended by allowing more than two vertices for edges.

Definition 2.2.4. A *Hypergraph* H is a pair (V, E) where V is a set of element called *vertices*, while E is a set of elements called *hyperedges*. A hyperedge is a non-empty set of vertices from V.

2.3 Query Analysis

After introducing our data sets, we will now start of our analysis. To start of, SPARQL queries are examined by looking at their used parts, features and their overall composition. This high-level analysis precedes a deeper structural analysis which will follow later.

2.3.1 Keywords and Operators

Multi-Source Collection A simple but effective first analysis to get an overview of queries is to look at the usage of keywords. The results for the Multi-Source Collection are shown in Table 2.3.

The table is divided into four blocks: types of queries, solution modifiers, SPARQL algebra operators, and aggregation operators. Each block is sorted by the number of occurrences in the *Valid* data set.

The first block in Table 2.3 describes the *type* of queries. In total, 91.96% (88.22%) are Select queries, 4.94% (3.38%) are Describe queries, 2.44% (6.56%) are Ask queries, and 2.44% (6.56%) are Construct queries. It is notable that there are tremendous differences between the data sets concerning the composition of query types. `BioMed13` has less than 3.47% (12.83%) Select queries and almost 94% (85%) Describe queries, whereas `LGD13` has 17% (28%) Select queries and almost 81% (71%) Construct queries.

The second block in Table 2.3 contains the usage of *solution modifiers*. Solution modifiers that are only used in minuscule numbers are omitted, in Table 2.3 this is the case for the Reduced solution modifier, it was only found in 6,126 (1,149) queries. Some notable observations from the data sets follow.

Almost all 89% (97%) of `BritM14` queries use Distinct. Similarly, but to a lesser extent, this is the case in in `BioP13` (96% (82%)) and `BioP14` (92%(68%)). In DBpedia there are significant differences: From '12 to '17, we have 21% (18%), 7% (8%), 16% (11%), 20% (38%), 6% (8%) and 26% (52%) of queries with Distinct respectively.

Limit is used most widely in `SWDF13` (48% (47%)), in `LGD13` (59% (17%)), and `LGD14` (54% (41%)). The most prevalent data sets for queries with Offset are `LGD14` (30% (38%)), `LGD13` (52%(13%)), and `DBpedia13` (10% (12%)).

Order By is used by far the most in Wikidata (44%), which may be due to the fact that `Wikidata17` is not a query log, but a wiki page that contains cherry-picked and user-submitted queries. These queries are intended to showcase system's behavior or highlight features of the Wikidata data set and should therefore produce a nice output. The other data sets are true query logs from query endpoints, which may therefore also contain the "development process" of queries: Users start by asking a query and gradually

Element	*#Valid*	*%Valid*	*#Unique*	*%Unique*
Select	311,496,923	91.96%	79,929,422	88.22%
Describe	16,727,191	4.94%	3,061,636	3.38%
Ask	8,265,673	2.44%	5,943,216	6.56%
Construct	2,255,793	0.67%	1,670,913	1.84%
Distinct	96,055,447	28.36%	29,973,911	33.08%
Limit	46,442,970	13.71%	17,043,706	18.81%
Offset	8,651,005	2.55%	4,112,839	4.54%
Order By	3,481,015	1.03%	1,609,921	1.78%
Filter	148,681,968	43.89%	34,609,372	38.20%
And	129,524,653	38.24%	26,737,378	29.51%
Optional	107,447,875	31.72%	13,119,429	14.48%
Union	85,024,759	25.10%	15,761,764	17.40%
Graph	27,556,055	8.13%	1,523,675	1.68%
Values	7,595,583	2.24%	5,086,033	5.61%
Not Exists	2,527,452	0.75%	1,096,099	1.21%
Minus	2,199,152	0.65%	1,664,359	1.84%
Exists	13,965	0.00%	7,832	0.01%
Group By	9,100,381	2.69%	3,887,216	4.29%
Count	924,474	0.27%	653,756	0.72%
Having	197,463	0.06%	40,401	0.04%
Avg	7,714	0.00%	731	0.00%
Min	7,040	0.00%	3,749	0.00%
Max	6,504	0.00%	3,796	0.00%
Sum	2,768	0.00%	785	0.00%

Table 2.3: Keyword count in Multi-Source Collection.

refine it until they have the one they want. This idea will be revisited in Section 5.1).

The third block has keywords associated to SPARQL algebra *operators* that occur in the body of queries. It can be observed Filter, And, Union, and Optional are quite common. Note that conjunctions in SPARQL can be denoted by "." or ";", or they can be implicit with the usage of groups, but for clarity they are grouped in And in Table 2.3. The next commonly used operator is Graph, but looking closer at the data, strikingly 96% (78%) and 85% (40%) of the queries using Graph originate from `BioP13` and `BioP14`. The use of Filter ranges from 63% (58%) for `DBpedia13` to 0.7% (3%) or less for `BioMed13` and `BioP13`, respectively.

Element	#O V	%O V	#R V	%R V
Select	664,323	99.83%	206,006,783	99.28%
Construct	516	0.08%	188,088	0.09%
Ask	306	0.05%	1,127,396	0.54%
Describe	327	0.05%	176,152	0.08%
Distinct	96,229	14.46%	15,848,797	7.64%
Limit	257,351	38.67%	38,149,526	18.39%
Offset	7,653	1.15%	13,878,747	6.69%
Order By	93,198	14.00%	18,172,260	8.76%
Filter	191,044	28.71%	36,835,864	17.75%
And	404,282	60.75%	73,920,238	35.62%
Optional	324,380	48.74%	31,402,334	15.13%
Union	61,246	9.20%	18,976,088	9.15%
Graph	6	0.00%	14	0.00%
Values	42,555	6.39%	66,432,384	32.02%
Minus	13,440	2.02%	1,781,620	0.86%
Not Exists	19,174	2.88%	412,645	0.20%
Exists	3,339	0.50%	95,218	0.05%
Group By	96,802	14.55%	825,957	0.40%
Having	6,084	0.91%	14,888	0.01%
Count	5,607	0.84%	36,215	0.02%
Avg	195	0.03%	1,505	0.00%
Min	2,101	0.32%	1,323	0.00%
Max	1,003	0.15%	376	0.00%
Sum	979	0.15%	684	0.00%

Table 2.4: Keyword usage in Wikidata Collection (Robotic (R) and Organic (O), *Valid* (V)).

The fourth block contains *aggregation* operators. It is surprising that these operators are used sparsely, even though aggregates are supported since SPARQL 1.1 (March 2013) [HS13]. In all data sets, each of these operators were used in 3% or less of the *Unique* queries, except for `LGD14` (31% with Count), `DBpedia17` (11% with Group By), and `Wikidata17` (30% with Group By). There is a higher relative use of aggregation operators in `Wikidata17` than in the other data sets, which is most likely due to the fact that `Wikidata17` is not a standard query log.

Element	#O U	%O U	#R U	%R U
Select	260,114	99.77%	34,261,882	99.23%
Construct	217	0.08%	24,587	0.07%
Ask	132	0.05%	73,019	0.21%
Describe	260	0.10%	167,563	0.49%
Distinct	50,401	19.33%	3,744,933	10.85%
Limit	72,247	27.71%	5,822,263	16.86%
Offset	2,979	1.14%	3,952,517	11.45%
Order By	45,918	17.61%	655,191	1.90%
Filter	93,494	35.86%	5,082,479	14.72%
And	156,911	60.18%	8,063,790	23.35%
Optional	99,855	38.30%	3,430,329	9.94%
Union	22,221	8.52%	852,956	2.47%
Graph	5	0.00%	4	0.00%
Values	11,887	4.56%	1,579,888	4.58%
Minus	6,951	2.67%	594,653	1.72%
Not Exists	8,281	3.18%	45,967	0.13%
Exists	2,068	0.79%	3,075	0.01%
Group By	35,746	13.71%	101,710	0.29%
Having	3,891	1.49%	1,162	0.00%
Count	1,647	0.63%	2,270	0.01%
Avg	133	0.05%	2	0.00%
Min	729	0.28%	60	0.00%
Max	388	0.15%	64	0.00%
Sum	348	0.13%	19	0.00%

Table 2.5: Keyword usage in Wikidata Collection (Robotic (R) and Organic (O), *Unique* (U)).

Wikidata Collection We now turn to Wikidata. Structured similarly to the table for the Multi-Source Collection, the keyword usage for the Multi-Source Collection is shown in Table 2.4 (*Valid*) and Table 2.5 (*Unique*).

Looking at the query types, over 99% (almost 100%, rounded up with 2 decimals) of queries are Select queries. This is true for robotic, organic, *Valid*, or *Unique* queries.

It is interesting to see that organic queries use the Distinct modifier around two times more often than robotic queries. Although Offset is almost not used in organic queries, they make sense for robotic queries to get more results are use 10 times more there.

As for operators, simple operators like Filter, And, and Optional are used in larger numbers in organic queries, Union is used almost the same, and Values is used much more often in robotic queries. Other operators are used very sparsely, in less than 1% of the cases. It is also interesting to note that Graph is almost never used.

The SPARQL algebra operators are almost not used, less than 1% of queries use them, with the notable exception of Group By in organic logs, which is used around 14%.

Note that a similar analysis on Wikidata logs was done in [BGK18] and in [Mal+18]. However, if you look closely at both papers side by side, you will see that there are small differences in their results. For example, the data set I1 is the same in both papers, as you can check by comparing Table 1 in [Mal+18] with the dates in Section 3 on Page 3 (under Table 2 and 3) in [BGK18]. Then, compare Table 6 in [BGK18] with Table 2 in [Mal+18]. You will see minor differences in each column for I1.

It is very easy to have minor differences in results for query logs of this large size. For instance, in our analyses, we noticed that just upgrading the SPARQL parser lead to such minor differences, which may very well explain what happened here. Also note that there are differences in how one can aggregate data. For example, there is a "Path with *" in their tables, whereas we considered our analyses for fragments. All in all, it makes sense that we present results that were gathered with identical methods for comparing the Wikidata Collection with Multi-Source Collection. After the above explanation, minor differences are expected, because they used a different parser (BlazeGraph as opposed to Jena).

2.3.2 Operator Set Distribution

We just looked at the keywords in keywords, but now we turn to the co-occurrence of operators inside the query bodies. This is important, because

Operator Set	*#Valid*	*%Valid*	*#Unique*	*%Unique*
none	107,285,016	33.32%	31,785,844	36.31%
A	15,106,778	4.69%	7,769,170	8.87%
F	30,679,572	9.53%	14,822,993	16.93%
A,F	9,583,490	2.98%	4,176,586	4.77%
CQF subtotal	162,654,856	50.51%	58,554,593	66.89%
O	2,921,810	0.91%	625,663	0.71%
A,O	3,436,987	1.07%	1,807,483	2.06%
F,O	7,115,439	2.21%	2,096,526	2.39%
A,F,O	24,512,799	7.61%	1,773,624	2.03%
CQF+O	+37,987,035	+11.80%	+6,303,296	+7.20%
U	8,533,645	2.65%	4,627,921	5.29%
A,U	1,627,742	0.51%	1,010,579	1.15%
F,U	627,559	0.19%	254,640	0.29%
A,F,U	1,824,697	0.57%	1,057,080	1.21%
CQF+U	+12,613,643	+3.92%	+6,950,220	+7.94%
V	151,078	0.05%	63,912	0.07%
A,V	207,180	0.06%	164,175	0.19%
F,V	2,497,572	0.78%	2,204,598	2.52%
A,F,V	142,211	0.04%	98,560	0.11%
CQF+V	+2,998,041	+0.93%	+2,531,245	+2.89%
G	26,288,960	8.16%	1,380,991	1.58%
A,G	391,433	0.12%	42,315	0.05%
F,G	876	0.00%	269	0.00%
A,F,G	34,418	0.01%	9,495	0.01%
CQF+G	+26,715,687	+8.30%	+1,433,070	+1.64%
A,F,O,U	67,026,601	20.81%	6,170,843	7.05%

Table 2.6: Sets of operators used in Multi-Source queries: And (A), Filter (F), Graph (G), Optional (O), Union (U), and Values (V).

this is required to partition queries into different fragments for analysis, which will be discussed in much more detail in Section 4.1.

Multi-Source Collection In Table 2.6, it can be seen that Filter, And, Union, Optional, and Graph are used fairly commonly in the bodies of Select, Ask, and Construct queries.

After looking at the individual operator occurrence in Section 2.3.1, it is

of interest to investigate how these operators occur together. In particular, the focus will be on queries for which the body *only* uses constructs with these operators. There is one exception: For `Wikidata17`, the Service subqueries before the analysis were removed. This was for approximately 200 queries. Wikidata has a special non-standard Service construct that is used to add labels to the output for human consumption, and this does not interact with other parts of the query.

The results are presented in Table 2.6, which has two kinds of rows. Each non-highlighted row has, in the left-most column, a set S of operators from $\mathcal{O} = \{\mathsf{Filter}, \mathsf{And}, \mathsf{Optional}, \mathsf{Graph}, \mathsf{Union}, \mathsf{Values}\}$, and in the rest of the columns, the amount of queries in the logs for which the body uses exactly the operators in S (and none from $\mathcal{O} \setminus S$). The value for *none* is the amount of queries that do not use any of the operators in $\mathcal{O}$ (including queries that do not have a body). The highlighted rows are special rows, as they contain the sum of a block, which can be the additional size gained from adding an operator to the $\mathsf{CQ_F}$ fragment. The columns are for absolute and relative numbers for the *Valid* and *Unique* data sets.

Conjunctive patterns with filters are considered to be an important fragment of SPARQL patterns, because they are believed to appear often in practice [NW10]; [Vid+10].

Definition 2.3.1. A *conjunctive query with filters ($\mathsf{CQ_F}$)* is a SPARQL pattern that only uses triple patterns and the operators And and Filter.

The Multi-Source Collection contains 50.51% (66.89%) $\mathsf{CQ_F}$ queries. Adding Optional to the $\mathsf{CQ_F}$ fragment would increase its relative size by 11.80% (7.20%) resulting in 62.31% (74.09%) of the queries. (Similarly for Union, Graph and Values.) Table 2.6 classifies 95.07% (96.62%) of the Select, Ask and Construct queries in the corpus. The remaining queries either use other combinations from $\mathcal{O}$ 1.64% (2.79%), or use other features than those in $\mathcal{O}$ in their body 2.10% (3.61%) like Bind, Minus, subqueries, or property paths. A recurring combination of features from $\mathcal{O}$ can be observed in the latest query logs (`DBpedia17`), in which Union and Values appear together in 1.30% (5.08%) of the queries, whereas they are mostly not existing in the other data sets.

Wikidata Collection Wikidata has a much higher occurrence of property paths than DBpedia logs, which is why the base fragment for further study of Wikidata logs will be looking at C2RPQs instead of CQs. This will be further explained in Section 4.1, for now it is sufficient to know that we are

now looking at results that include property paths as opposed to excluding them like before when looking at the Multi-Source Collection.

Like before, we look at the basic operators And, Filter, Optional, and Union. But we add Values, Bind, and Service. The reason for choosing these operators will become clearer shortly, when you look at the operators that are present in the logs. These operators are chosen for subsequent analyses, but the reason for this is derived from the results here.

Before we turn to the results, consider this: With these seven operators, this allows for 128 combinations of operators. Interestingly, almost all combinations occurred at least once in the logs, with only 8 combinations missing. In contrast to the DBpedia logs, the Graph operator is not included as it was only in 20 queries, so it does not warrant including it; doing so would also increase the number of possible combinations to 256.

The occurrences of operator combinations in Wikidata Collection can be seen in Table 2.7 and Table 2.8 (for *Valid* and *Unique*, resp.). The table is structured to show the gains of adding single or small sets of operators to CQs. Specifically, the designation X was chosen to represent the extended operators Service, Bind, and Values as operator set. This is done for space and clarity, showing 26 instead of 256 rows, so it still keeps the presentation similar to Table 2.6. It is still important to note that the table does not give a complete picture. This can be surmised when looking at the low percentages in the table. For Wikidata, operators are much more diversified than in DBpedia logs, as they use a more operators apart from the basic ones, and there are more combinations.

The parts omitted from the table are mostly not interesting and only show rare combinations with less than 1%, but there are some exceptions. These interesting observations are reported now:

- There are some single operator combinations that stand out. Optional and Service, and both of them with And amount to 11-12% of organic *Valid* queries. The reason for this is most likely that Service is very vital for Wikidata because of the usage of the special labeling service. This is explained in Section 4.1.2.

- 31% of robotic *Valid* queries use only Values without any other operators. This shrinks to less than 3% when going to *Unique* queries, so these are duplicate queries.

- Although it does not have a keyword and may be easily overlooked, the *Subquery* operator has a non-negligible occurrence in organic queries, it must be treated like other operators, therefore removing queries from the operator combinations under scrutiny.

Operators	#O V	%O V	#R V	%R V
none	53,980	8.12%	54,402,277	26.24%
A	33,076	4.97%	29,092,643	14.03%
F	6,056	0.91%	4,055,489	1.96%
A,F	25,235	3.79%	5,807,108	2.80%
C2RPQ+F	118,347	17.79%	93,357,517	45.03%
O	4,953	0.74%	233,869	0.11%
A,O	4,481	0.67%	801,598	0.39%
F,O	7,876	1.18%	1,022,353	0.49%
A,F,O	11,633	1.75%	5,058,602	2.44%
C2RPQ+F+O	+28,943	+4.35%	+7,116,422	+3.43%
U	681	0.10%	66,365	0.03%
A,U	1,336	0.20%	86,483	0.04%
F,U	27	0.00%	19,893	0.01%
A,F,U	2,223	0.33%	167,385	0.08%
C2RPQ+F+U	+4,267	+0.64%	+340,126	+0.16%
S	29,696	4.46%	6,061,738	2.92%
A,S	48,687	7.32%	1,759,288	0.85%
F,S	5,913	0.89%	158,291	0.08%
A,F,S	11,711	1.76%	57,230	0.03%
C2RPQ+F+S	+96,007	+14.43%	+8,036,547	+3.88%
X	45,225	6.80%	71,794,220	34.63%
A,X	58,082	8.73%	6,476,686	3.12%
F,X	6,790	1.02%	191,940	0.09%
A,F,X	18,235	2.74%	797,879	0.38%
C2RPQ+F+X	+128,332	+19.29%	+79,260,725	+38.23%
A,F,O,U	685	0.10%	8,454	0.00%

Table 2.7: Sets of operators used in Wikidata queries: And (A), Filter (F), Optional (O), Union (U), Service (S), and combinations of Values and Service and Bind (X), (Robotic (R) and Organic (O), *Valid* (V)).

Operators	#O U	%O U	#R U	%R U
none	23,132	3.48%	20,540,542	9.91%
A	14,018	0.01%	2,490,457	1.20%
F	3,855	0.00%	2,535,408	1.22%
A,F	13,473	0.01%	197,751	0.10%
C2RPQ+F	54,478	3.49%	25,764,158	12.43%
O	3,139	0.47%	196,369	0.09%
A,O	2,061	0.31%	215,933	0.10%
F,O	5,716	0.86%	106,060	0.05%
A,F,O	5,505	0.83%	891,591	0.43%
C2RPQ+F+O	16,421	2.47%	1,409,953	0.68%
U	219	0.03%	29,431	0.01%
A,U	549	0.08%	12,685	0.01%
F,U	21	0.00%	1,065	0.00%
A,F,U	1,045	0.16%	7,043	0.00%
C2RPQ+F+U	1,834	0.28%	50,224	0.02%
S	17,105	2.57%	1,054,858	0.51%
A,S	25,460	3.83%	481,047	0.23%
F,S	1,255	0.19%	16,474	0.01%
A,F,S	5,800	0.87%	12,399	0.01%
C2RPQ+F+S	49,620	7.46%	1,564,778	0.75%
X	24,924	3.75%	2,187,076	1.05%
A,X	30,049	4.52%	1,187,355	0.57%
F,X	1,606	0.24%	36,565	0.02%
A,F,X	8,987	1.35%	320,375	0.15%
C2RPQ+F+X	65,566	9.86%	3,731,371	1.80%
A,F,O,U	359	0.05%	487	0.00%

Table 2.8: Sets of operators used in Wikidata queries: And (A), Filter (F), Optional (O), Union (U), Service (S), and combinations of Values and Service and Bind (X), (Robotic (R) and Organic (O), *Unique* (U)).

Table 2.7 shows that just adding Union has a very small impact. However, there are some operators that have a larger impact. For one, Service adds over 14% of organic queries in the *Valid* logs. This is not true for robotic logs. However, also including operators Values and Bind in addition to Service adds 38% of robotic queries in *Valid* logs. This is mostly due to the huge occurrence of Values in robotic logs.

So overall, this shows why adding Service for organic queries and Values for robotic queries is essential. With this, most of the Wikidata query logs are covered, which would not be the case otherwise.

As already noted at the end of Section 2.3.1 when discussing the keywords in Wikidata, there is a similar analysis in [BGK18] (Table 7) and [Mal+18] (Table 3). In contrast to their keyword analysis, the tables for co-occurrences are not directly comparable, because they lack absolute numbers and the numbers are aggregated differently (organic and robotic are split into subsets based on time). However, as we discussed in the keyword analysis, we noted that their results are not completely identical down to the last digit. For the same reasons that we discussed, it makes sense to present our results for the Wikidata Collection to compare them with the Multi-Source Collection. Additionally, there are many differences in the way the results can be aggregated to effectively present data, which was discussed before (e.g. using an operator set for Service, Bind, and Values).

2.3.3 Query Sizes and Other Measures

Triples in Multi-Source Collection In order to measure the size of the queries in the data sets, the total number of triples of the kind $\langle s, p, o \rangle$ contained in Select, Ask and Construct queries were counted. For this, merely the number of triples contained in each query were counted without further investigating the possible relationships among them (such as join conditions, unions etc.), which are studied later. Only Select, Ask and Construct queries are considered, while Describe statements are discarded, because they have implementation-dependent semantics. And the Describe statements in the corpus are vastly different to the remaining query types: 95% (97%) of them do not have a body and therefore no triples.

Figure 2.1 illustrates how queries containing 0 to 11+ triples are distributed relatively to their total size over Select, Ask and Construct queries in each of the data sets.

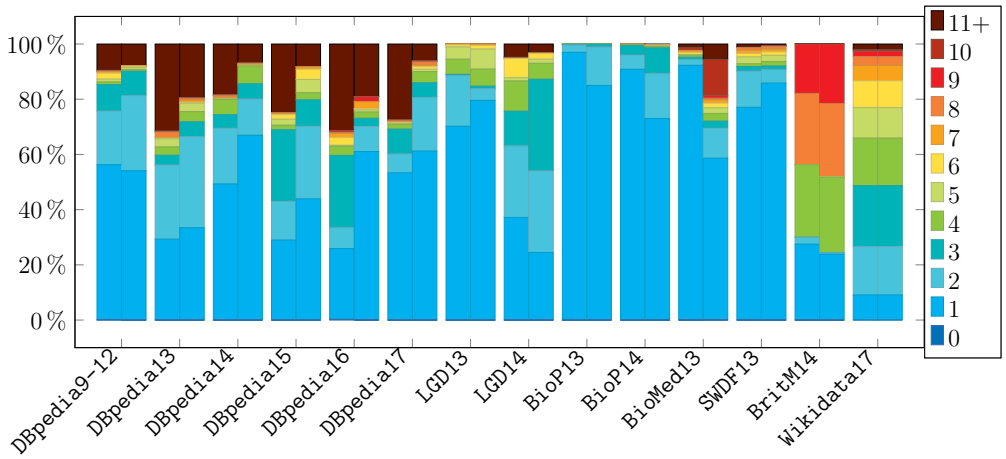


Figure 2.1: Percentages of queries exhibiting different number of triples (in colors) for each data set for Valid (left hand side of each bar) and Unique queries (right hand side of each bar).

A first observation that can draw be drawn from the figure is that for the majority of the data sets, the queries with a low number of triples (from 0 to 2) have a noticeable share within the total amount of queries per data set. Whereas these queries are almost the only queries present in the `BioP13` and `BioP14` data sets, they have the least concentration in `BritM14` and `Wikidata17`. The rest of the data sets have unique characteristics, `BritM14` being a collection of queries with fixed templates and `Wikidata17` being the most diverse data set of all, gathering queries of rather disparate nature that are representatives of classes of real queries issued on Wikidata. Finally, `DBpedia9-12` until `DBpedia17`, along with `LGD14` and `BioMed13` are the data sets exhibiting the most complex queries with extremely high numbers of triples exceeding 10. Note that `BioMed13` has almost 94% (87%) Describe queries. The numbers reported here only depict the remaining 6% (13%).

Overall, it can be observed that 63.62% (58.40%) of the Select, Ask and Construct queries in the corpus use at most one triple, 77.89% (90.16%) uses at most six triples, and 99.44% (98.35%) at most twelve triples. The largest queries we found came from `DBpedia15` (209 and 211 triples) and `BioMed13` (221 and 229 triples). In the query logs of `DBpedia17`, the largest queries contain 207 and 209 triples.

Triples and Symbols in Wikidata Collection Figure 2.2 reports the distribution of length of the queries in terms of the number of their triples. Contrarily to the triple count in Figure 2.1 for the Multi-Source Collection, property paths are included in this metric. The reason is that a property

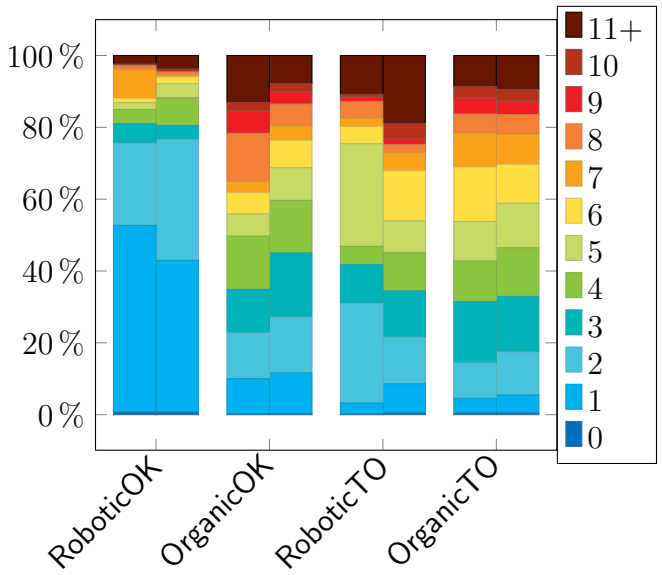


Figure 2.2: Percentages of queries with corresponding sizes (number of triples *plus sizes of property paths*) for each data set— Valid (left) versus Unique (right). The sizes are reported on the left-hand side and range from 0 to ≥ 11.

path pattern of the form `?x wdt:P31/wdt:P279 ?y` is a shorthand notation for *two* triples and we feel that it should be considered as such. We therefore count property path patterns as follows. Let *pp* be a property path encoded as a regular expression. We say that the *size* of *pp* is the number of alphabet symbols in the regular expression. For example, the property paths wdt:P279/wdt:P279 and wdt:P279/wdt:P279* have size two. The query illustrated in Listing 2.2 has size four: it contains two triples of size one and one triple of size two.[6] We analyzed the triples and property paths inside the Select, Ask, and Construct clauses and explicitly exclude the queries with a Describe clause. The results for the four data sets can be found in Figure 2.2. Here, each bar is split into two sides, where the left side represents the *Valid*, and the right side the *Unique* version of each data set.

The conclusions of this specific analysis are similar to those made for the Wikidata Collection, assessing that user queries are more diverse than robotic ones. Nevertheless, the triple count distributions shown in Figure 2.2 shows the breakdown in terms of the other dimensions considered in our analysis, encompassing the *Valid* and *Unique* queries and the OK and timeout

[6]Notice that the query size computed as described above even with Kleene-star ('*') and transitive closure operators ('+') does not depend on the length of the actual paths in the graph instance when evaluating the query.

#const	*#Organic*	*#Robotic*	*#var*	*#Organic*	*#Robotic*
0–9	239,003	33,485,122	0–9	255,383	33,654,282
10–19	21,307	480,936	10–19	3,999	423,415
20–32	153	393,430	20–27	1,081	281,791

Table 2.9: Number of constants and variables in the triples in the Wikidata Collection.

queries. The timeout queries for instance show a relatively higher complexity in terms of number of triples than the OK ones, and this observation also applies to RoboticTO queries, that are thus quite different from RoboticOK queries. This seems to suggest that timeout queries are queries that failed because of a higher number of triples, which could be interesting to consider in graph query evaluation and optimization studies. The information about the average number of triples confirmed this, since *Valid* queries have on average 2.58 (2.65) triples, whereas timeout queries have 5.65 (5.94) triples. As a side remark, the highest number of triples that we observed is in the RoboticOK *Valid* logs and is equal to 67, which was found in 68 queries (in 34 queries in the RoboticOK Valid logs, respectively). The largest size of a property path triple was 19.

Number of Constants and Variables in Wikidata Collection Counting the number of triples is only one possible measure for the complexity of a query. We enrich this analysis by considering further characteristics of the triples, i.e., whether the triples contain variables or constants. This information is useful for the shape analysis that we conduct in Section 4.2.1, in which we show actual differentiations in the obtained shapes by removing or including the constants. We only count the number of *distinct* variables and constants in these experiments. For the Wikidata Collection, Table 2.9 reports the numbers of constants and variables by intervals in the *Unique* OrganicOK and RoboticOK logs.

Precisely, we counted the number of different variables (different constants, respectively) of each query in the logs and reported the total number of queries that have this number. For conciseness, these numbers are aggregated into intervals in Table 2.9. We can observe that for numbers (of constants and variables) greater than 11, the queries with these numbers of *constants* are more abundant than queries with these numbers of variables.

#const	#Valid	#Unique	#var	#Valid	#Unique
0-9	254,233,181	81,120,041	0-9	310,909,696	86,331,142
10-19	67,397,924	6,360,502	10-19	10,755,144	1,165,931
20-29	352,827	54,939	20-29	340,157	43,470
30-39	28,383	6,830	30-39	7,842	1,735
40-49	1,098	700	40-49	770	654
50-59	4,595	335	50-59	729	306
60-69	182	121	60-69	3,798	99
70-69	55	37	70-69	158	141
80-89	87	26	80-89	46	39
90-99	30	11	90-99	22	22
100-109	9	3	100-109	25	16
110-130	16	4	110-130	2	1
130-231	2	2			

Table 2.10: Number of constants and variables in the triples in the Multi-Source Collection.

Constants and Variables in Multi-Source Collection Table 2.10 shows the usage of constants and variables in the Multi-Source Collection. We first examine the constants. Most shapes use less than 10 constants. For the next incremental step, more than 3 times less queries use less than 20 constants. This is even more drastic if one looks at unique queries, then more than 10 times use less than 20 constants. This supports the statement that queries are small in general.

If one looks at variables as opposed to constants, then even more use only less than 10 variables and more than 30 times less use less than 20. It seems natural, that more queries use variables, because it stands to reason that most users want more than yes or no answers from queries, which is also supported by the number of Select queries compared to the number of Ask queries. One final staggering observation: Going to *Unique* logs, more than 74 times less queries use less than 20 variables. This means that queries with less than 10 variables absolutely dominate, which may have been expected when looking at earlier results about size.

Even though most queries use less than 10 constants or variables, there are a few queries that use a large number. For constants, this goes up to 231, while for variables, this goes up to 130. However, these top counts are only for 1-2 unique queries.

2.3.4 Query Features: Subqueries and Projection

We will now look at two features that do not fit in the previous discussed categories.

Projection plays a crucial role in the complexity of query evaluation. Many papers [BPS15]; [KK16]; [Let+13]; [PAG09]; [PV11] define evaluation as the following question: *Given an RDF graph G, a SPARQL pattern P, and a mapping μ, is μ an answer to P when evaluated on G?* In other words, the question is to verify if a candidate answer μ is indeed an answer to the query. Projection refers to discarding attributes of a query result. If P is a CQ, this problem is NP-complete if the queries use projection [BPS15]; [CM77]; [Let+13], but its complexity drops to PTIME if projection is absent [BPS15]; [Let+13]; [PAG09]. This difference can be understood as follows: If the query tests the presence of a k-clique, then without projection we are given a k-tuple of nodes and need to verify if they form a k-clique. With projection, we need to solve the NP-complete k-clique problem, because we are not given back the complete k-clique to just verify. Therefore, the use of projection has a huge influence of the complexity of query evaluation.

Multi-Source Collection Only 1,309,040 (575,666) queries in our corpus use subqueries. The feature was most used in `Wikidata17` (9.74%), about an order of magnitude more than in any of the other data sets.

Surprisingly, with the Multi-Source Collection it turned out that at least 9.1% (13.13%) of the queries use projection, which is significantly higher than what Picalausa and Vansummeren discovered in DBpedia logs from 2010 [PV11].

The numbers for projection can be split by query type, either Select or Ask queries are eligible for projection. The 9.1% (13.13%) consists of 8.33% (11.88%) Select queries plus 0.76% (1.24%) Ask queries. Notice that the total number of Ask queries 2.44% (6.56%) is significantly higher, even though they just return a Boolean value, and one would intuitively expect that almost all of them would use projection. The reason is that most Ask queries do not use variables: They ask if a *concrete* RDF triple is present in the data. Following the test for projection in Section 18.2.1 of the SPARQL W3C Recommendation [HS13], these queries were classified as not using projection.

As for the total projection, due to the use of the Bind operator, or to the presence of subqueries, there was a number of queries (3.08% for *Valid* and 5.37% for *Unique* queries) for which no definite determination could be made if they use projection or not without significantly complicating the notion and test for projection. Therefore the number of queries with projection lies between 9.1% (13.13%) and 12.18% (18.5%) for all queries.

Wikidata Collection Roughly 1% of the queries in the *Unique* Logs (including RoboticOK and OrganicOK queries) use *subqueries*. This number goes down to 0.37% for the corresponding subqueries in the *Valid* logs.

We also ran a test for mining the number of queries that use projection. As already explained, projection is a cause of complexity increase of query evaluation for CQ queries, that goes from NP-complete is projection is present to PTIME if projection is absent [BPS15]; [Let+13]. As before, we use the test for projection in accordance to in Section 18.2.1 in the SPARQL 1.1 recommendation [HS13] Out of the valid queries we found 25,569,947 (12.28%) queries that use projection.[7] Out of the valid Organic queries the amount of queries with projection even rises to 28.85%. These percentages become 18.01% and 28.05% for the unique valid and Organic queries, respectively. In particular, they are much higher than the 13.12% Select/Ask queries found within the *Unique* query logs of the DBpedia corpus in [BMT17a].

[7]This is a lower bound, since our test is sound but incomplete.

3

Properties and Property Paths

The predicate of SPARQL triples has a special status. The predicate is the verb of a semantic triple, which gives relations in an ontology a specialized meaning, as they define properties of objects. In contrast to subjects and objects, which are only simple variables or constants, predicates can be property paths, which are essentially a special flavor of regular expressions. In the following sections, the predicate position is examined in depth.

3.1 Usage of Wikidata Properties

Before turning to property paths, since Wikidata Collection is a data set from a single source, it was interesting to look at the general usage of properties. The OrganicOK *Unique* query logs were chosen, because on the surface, this set seems the most suitable, since the query endpoints are meant to be use facing.

Properties are IRIs, but the prefixes are shared, so the final path element is the main identifier (P856, P31,...). The total number of properties found in these logs are 881,490. These are divided into two major namespaces at a top level: `www.wikidata.org` (805,196), and `www.w3.org` (70,829).

Figure 3.1 is a sunburst diagram showing the segmentation of the Wikidata properties inside the largest top-level namespace qualifier `www.wikidata.org`. In order to avoid clutter and for ease of presentation, the sunburst is solely annotated with the properties with occurrences that are above a given threshold (6,000).

More views about the sunburst including the other top-level properties and with complete information about all the number of occurrences, is available via an interactive version of the diagram [Tim19b]. Hovering over the various segments of the rings provides the information omitted here in order

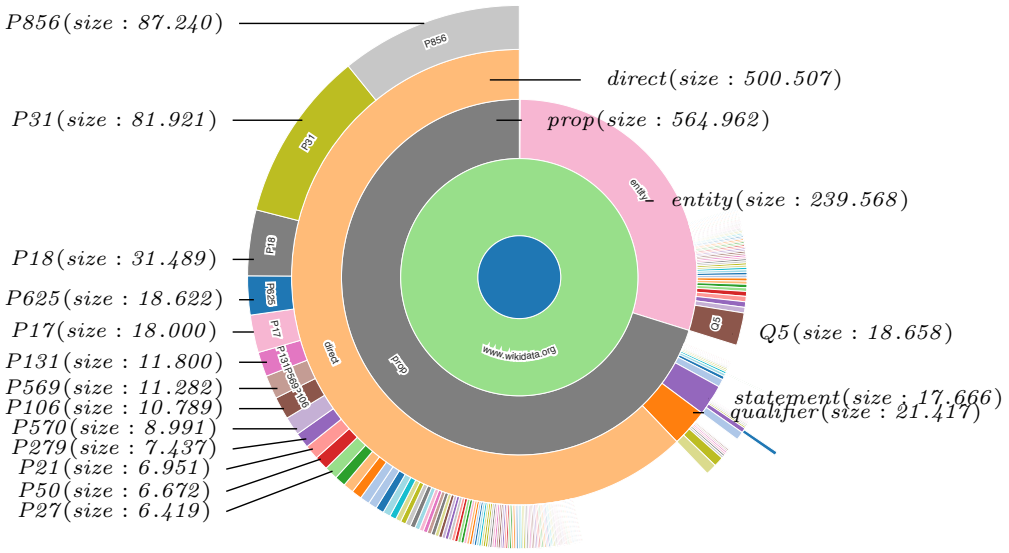


Figure 3.1: Sunburst distribution of the property qualifiers in Wikidata queries (Unique OrganicOK query logs). Interactive version available at https://podmr.github.io/darql/property-sunburst/.

ID	*Title*
P17	country
P18	image
P21	sex/gender
P27	country of citizenship
P31	instance of
P50	author
P106	occupation
P131	located in the administrative territorial entity
P279	subclass of
P569	date of birth
P570	date of death
P625	coordinate location
P856	official website

Table 3.1: Most common properties in Wikidata and their labels.

to avoid clutter in Figure 3.1.

3.2 Property Path Usage in Multi-Source Collection

In the Multi-Source Collection a total of 1,412,762 (329,984) queries used property paths. From these queries, 1,528,701 (404,721) property paths were extracted in total. Although property paths are therefore rare, this is not so for every data set: 92 queries (29.87%) in `Wikidata17` have property paths.

A large fraction of these property paths are extremely simple. For instance, 65,693 (63,428) property paths are $!a$ ("follow an edge not labeled a"), and 80,421 (58,156) are $\hat{}a$ ("follow an a-edge in reverse direction").

In total, 65,751 (63,478) queries use the *different-from* operator "!", and 394,726 (144,569) use the reverse navigation operator "$\hat{}$".

In Table 3.2, an overview of all property paths found in the corpus is presented. For readability purposes, the concatenation operator "/" of SPARQL is not explicitly denoted, i.e. it is elided, so a/b is written simply as ab infix. Furthermore, in the classification, $\hat{}a$ and $!a$ are treated as the same as a literal. For instance, ab, $(\hat{}a)b$, and $(!a)b$ are all classified as $a_1 \cdots a_k$ with $k = 2$. Capital letters are used to to denote subexpressions that can match a set of different IRIs. For example, $(a|b)$ can match a and b, i.e., a set of two symbols. The sizes of theses sets is noted in the column *Set Sizes*.

If the expression uses the !-operator, it can actually be matched by an infinite number of IRIs and can be seen as a wildcard test. (Some users even write the expression $(!a|!b)$ to obtain a wildcard that can match any IRI.) If an expressions uses the !-operator, it is annotated this with (wc) denoting a wildcard in the *Set Sizes* column.

Furthermore, each row represents the expression type listed on the left plus its symmetric form. For instance, a^*b counts expressions of the form a^*b and ba^*. The variant listed in the table is the one that occurred most often in the data. That is, a^*b occurred more often than ba^*.

Bagan et al. [BBG13] proved a dichotomy on the data complexity of evaluating property paths under *simple path* semantics, i.e., expressions can only be matched on paths in the RDF graph in which nodes appear only once. They showed that, although evaluating property paths under this semantics is NP-complete in general, it is possible in Ptime if the expressions belong to a class called C_{tract}. Remarkably, only eight expressions were found in the corpus which are not in C_{tract}, namely $(ab)^*$ (once) and $ab(ab)^*$ (seven times). The complexity of enumerating answers to property paths of the form as in Table 3.2 is studied in [MT17]. More precisely, the paper investigates enumeration problems for *simple transitive expressions*, which capture 394,726 (144,569) of the expressions in Table 3.2.

Expression	*#Valid*	*%Valid*	*#Unique*	*%Unique*	*Set Size*	*k*
a^+	618,459	40.46%	5,968	1.47%		
A^*	361,402	23.64%	89,379	22.08%	≤ 4 (wc)	
a^*	160,628	10.51%	68,681	16.97%		
a^*b	23,523	1.54%	20,566	5.08%		
a^*b^*	14,674	0.96%	997	0.25%		
$A^*B?$	7,252	0.47%	1,326	0.33%	≤ 5	
abc^*	70	0.00%	54	0.01%		
$(ab^*)\|c$	45	0.00%	15	0.00%		
$a^*b?$	45	0.00%	15	0.00%		
A^+	19	0.00%	18	0.00%	≤ 7 (wc)	
$ab(ab)^*$	7	0.00%	7	0.00%		
$a^+\|b^+$	3	0.00%	3	0.00%		
Ab^*	2	0.00%	1	0.00%	≤ 1 (wc)	
aB^*	2	0.00%	2	0.00%	≤ 2 (wc)	
$a\|b^*$	2	0.00%	2	0.00%		
$a\|b^+$	2	0.00%	2	0.00%		
$A^+B?$	1	0.00%	1	0.00%	≤ 5	
A^*B	1	0.00%	1	0.00%	≤ 5	
A^*bc	1	0.00%	1	0.00%	$= 5$	
$a?b^*$	1	0.00%	1	0.00%		
$(ab)^*$	1	0.00%	1	0.00%		
A	139,662	9.14%	129,515	32.00%	≤ 6 (wc)	
$a_1 \cdots a_k$	109,166	7.14%	25,431	6.28%		≤ 6
$\hat{}a$	80,421	5.26%	58,156	14.37%		
$a?$	9,864	0.65%	3,347	0.83%		
$a_1? \cdots a_k?$	2,704	0.18%	971	0.24%		≤ 5
$a_1? \cdots a_{k-1}?a_k$	664	0.04%	197	0.05%		≤ 3
$aB?$	40	0.00%	34	0.01%	≤ 2	
$ab?c?d$	12	0.00%	10	0.00%		
Ab	8	0.00%	6	0.00%	≤ 2	
AB	7	0.00%	4	0.00%	≤ 2	
$a\|ba\|c\|d$	6	0.00%	2	0.00%		
$A?$	4	0.00%	4	0.00%	≤ 2 (wc)	
$abc?d?$	2	0.00%	2	0.00%		
$AAAAAA$	1	0.00%	1	0.00%	$= 2$	
Total	1,528,701	100%	404,721	100%		

Table 3.2: Structure of property paths in Multi-Source Collection. Capital letters denote unions of symbols or wildcards (wc). Transitive expressions are on top, non-transitive on bottom.

3.3 Property Path Usage in Wikidata Collection

Property paths are much more common in the Wikidata Collection than in the Multi-Source Collection. Also, since their origin is known, the analysis can be split into two parts, which can be quite interesting for comparison.

The main results are summarized in Table 3.3 and Table 3.4 for Robotic queries and Organic respectively.

Overall, 49,971,258 (13,480,433) queries in these logs use property paths, which amounts to a total of 24.03% (38.94%) of the entire logs. In these queries 165,343 (82,764) property paths are in Organic queries, and 55,168,101 (14,106,489) in Robotic ones. Note that a query can contain more than one property path, the numbers reported just report the presence in queries.

The number of queries in the Wikidata Collection is significantly larger (57x) than in the Multi-Source Collection.

Both Organic and Robotic property path log sets are interesting for an analytical study and deserve a deeper inspection in order to classify the occurring path expressions into distinct types. Indeed, when looking at the *structure* of these path expressions, there are 234 different types of Organic expressions, compared to only 64 types of Robotic expressions. This kind of thorough classification revealed the different characteristics of the organic property paths with respect to the robotic ones, as the former exhibit more variety and heterogeneity than the latter despite their lower occurrences. The *type* of a property path is obtained as follows. Each variable or IRI is replaced by letters from the alphabet in increasing order. (If a variable or IRI is repeated in the property path, it is replaced by the same alphabet letter.) For example, `wdt:P31`*`/wdt:P279`* is of the type a^*b^* and `wdt:P31/wdt:P31`*`/wdt:P279`* is of the type aa^*b^*.

Robotic Property Paths Table 3.3 contains a summary of the most common types of property paths in robotic queries. The columns with "V" represent results for the *Valid* queries, and the columns with "U" for the *Unique* queries. For succinctness, different types are aggregated together. For example, each type is aggregated with its reverse type. For instance, the row for ab^* also contains the expressions of the form a^*b. Furthermore, $\hat{}a$ ("follow an a-edge in reverse direction") is treated the same as a single label.[1] Finally, disjunctions are also grouped together, denoted by capital letters. In

[1]The operator $\hat{}$ is used in 0.80% (1.10%) of robotic and 2.03% (3.18%) of organic queries.

Expression Type	*#Valid*	*%Valid*	*#Unique*	*%Unique*
a^*	27,850,487	50.48%	1,392,865	9.87%
ab^*, a^+	9,417,166	17.07%	2,816,134	19.96%
ab^*c^*	823,153	1.49%	67,502	0.48%
A^*	328,895	0.60%	51,860	0.37%
ab^*c	122,286	0.22%	1,680	0.01%
a^*b^*	62,784	0.11%	608	0.00%
abc^*	27,287	0.05%	4,083	0.03%
$a?b^*$	15,893	0.03%	11,999	0.09%
A^+	4,674	0.01%	2,043	0.01%
Ab^*	1,562	0.00%	674	0.00%
Other transitive	1,643	0.00%	161	0.00%
$a_1 \cdots a_k$	13,382,005	24.26%	9,368,442	66.41%
A	3,043,725	5.52%	381,434	2.70%
$A?$	31,150	0.06%	296	0.00%
$a_1a_2? \cdots a_k?$	25,872	0.05%	5,940	0.04%
$\hat{a}$	21,202	0.04%	471	0.00%
$abc?$	7,620	0.01%	8	0.00%
Other non-transitive	697	0.00%	289	0.00%
Total	55,168,101	100%	14,106,489	100%

Table 3.3: Structure of property paths for all robotic Wikidata queries.

Table 3.3, a capital letter A denotes a subexpression that matches a *disjunction of at least two symbols*. Empirically, an A either denotes an expression of the form $!a$, $(a|!a)$, or a disjunction of the form $(a_1| \cdots |a_k)$ with $k > 1$. Table 3.3 is divided into *transitive expressions* (top) and *non-transitive expressions* (bottom). Transitive expressions are those that match arbitrarily long paths (i.e., they use the operators * or $^+$). The empty cells represent values that round down to 0.00%.

Interestingly, there are significant differences between the numbers of expressions in the *Valid* and in the *Unique* sets. Whereas the type a^* accounts for 50.48% of the expressions in the *Valid* data set, this drops to 9.87% in the *Unique* set. On the other hand, concatenations of symbols (type $a_1 \cdots a_k$) represent 24.26% in the valid queries, but over 66% in the unique queries. To further understand this phenomenon, the following was done. Taking the RoboticOK and OrganicOK, the most popular 20 queries (based on occurrence) containing a property path with Kleene-star were computed.

The top most popular robotic query with property paths (having 281, 096

Expression Type	*#Valid*	*%Valid*	*#Unique*	*%Unique*
AB^*	57,913	35.03%	28,034	33.87%
A^*	41,777	25.27%	22,071	26.67%
ABC^*	6,497	3.93%	3,044	3.68%
a^*b^*	3,330	2.01%	849	1.03%
ab^*c^*	2,704	1.64%	1,172	1.42%
$a^*B_1?b_2? \cdots b_k?$	1,789	1.08%	422	0.51%
$ab\|c^*d$	1,514	0.92%	534	0.65%
$a^*\|b^*$	347	0.21%	253	0.31%
$abCD^*$	283	0.17%	219	0.26%
ab^*c	113	0.07%	90	0.11%
$a^*\|B$	102	0.06%	76	0.09%
$\sim (ab)^*$	101	0.06%	82	0.10%
ab^*c^*d	86	0.05%	72	0.09%
$ab^*\|c$	70	0.04%	59	0.07%
$a^*b?c$	56	0.03%	27	0.03%
$a^*b^*c^*$	32	0.02%	28	0.03%
$ab^*\|b^+a^*$	16	0.01%	12	0.01%
ab^+c	13	0.01%	12	0.01%
$ab^*\|cd^*$	13	0.01%	12	0.01%
a^*bc^*	11	0.01%	11	0.01%
Other transitive	22	0.01%	20	0.02%
$A_1a_2 \cdots a_k$	31,032	18.77%	15,754	19.03%
A	13,248	8.01%	7,592	9.17%
$a_1? \cdots a_k?$	1,938	1.17%	1,470	1.78%
$A?$	1,178	0.71%	302	0.36%
$ab_1? \cdots b_k?$	1,117	0.68%	529	0.64%
$ab\|c$	27	0.02%	5	0.01%
Other non-transitive	14	0.01%	13	0.02%
total	165,343	100.00%	82,764	100.00%

Table 3.4: Structure of property paths for all organic Wikidata queries.

occurrences) belong to the second most occurring type in Table 3.3 since it contains a single path expression of the kind `wdt:P31/wdt:P279`*. The query is in fact a *conjunctive regular path query*. By looking at the top most popular organic query (with $1,778$ occurrences), we can see that it belongs to the first most occurring type in Table 3.4 and exhibits the same single path expression. As a side remark, the latter query is fairly more complex and uses Union, Filter, Bind, and Service clauses.

Organic Property Paths Table 3.4 contains results on the Organic data sets. Since there are 234 different types of expressions, the results needed to be aggregated more aggressively to make the results presentable.

The types are grouped into 41 different categories, from which some in the table are omitted. The main difference with Table 3.3 is that a capital letter is allowed to denote a single symbol. So, A can denote expressions equivalent to a, $(a_1|\cdots|a_k)$, $!a$, or $(!a|a)$. The other difference is that the types $(ab)^*$ and $a(bc)^*$ are grouped together in the type $\sim(ab)^*$. These expressions stand out from the rest, since they are the *only* type of transitive expressions that put length constraints on arbitrarily long paths. Indeed, all other transitive expressions allow paths of arbitrary length once the length exceeds a certain value. This is not the case for $(ab)^*$, since it only allows paths of even length.

Here, the percentages in the unique sets are quite similar to those in the valid sets. The Organic queries generally contain more challenging property paths to valuate. On average, organic property paths are also larger than robotic ones. They contain 2.07 (2.01) literals on average, whereas robotic property paths only contain 1.49 (1.89) literals on average. There were even expression types which occurred more often in the TO (timeout) logs than in the OK logs, such as $\sim(ab)^*$. This is interesting, because such expressions are known to be complex (NP-complete) to evaluate under simple path semantics [BBG13].

Additional Insights on Wikidata Property Paths In this section, the differences between Table 3.3 and Table 3.4 and the results of the property paths analysis of the Multi-Source Collection are discussed. The remarkable difference between the Wikidata Collection and the former (which was done on a corpus mainly consisting of DBpedia queries) is that here, a much larger fraction of the queries use property paths. This is probably due to the peculiar characteristics of the Wikidata data. Property paths are often used in queries performing class navigation in Wikidata and emulating ontological reasoning. Wikidata has relatively long paths in the data that are labeled with the same label (and that are popular to query, e.g., InstanceOf paths), whereas DBpedia has comparably shorter paths and is more flat.

The remarkable similarity, however, is in the *structure* of the property paths. Martens and Trautner [MT18a] defined the class of *simple transitive expressions*, which are syntactically very restricted, but covered over 99% of the property paths in the Multi-Source Collection. In the Wikidata Collection, 1.61% (0.48%) of the Robotic and 3.83% (2.72%) of the Organic property paths are not simple transitive expressions. The most significant reason why property paths fall out of this fragment is the use of a^*b^* as a subexpression,

whereas simple transitive expressions only use one subexpression with Kleene star.

Furthermore, all property paths except 198 (98) are in $\mathsf{C}_{\mathsf{tract}}$, which is a broader class introduced by Bagan et al. [BBG13] and which precisely characterizes the set of regular expressions for which the data complexity under simple path semantics is tractable if P $\neq$ NP.

3.4 Tree Pattern Queries

Tree pattern queries (e.g., [Cze+15]; [Cze+18]; [KS08]; [MS04]) are a well-studied query formalism on trees which is inspired on XPath but which can just as well be used for querying graph-structured data [Cze+18]; [LMV16a]. We next define a *tree-pattern-like* fragment of our queries and investigate how common it appears in the logs.

Property paths have the power to do *forward* and *backward* navigation through edges. For instance, if a is an IRI, then the property path $\hat{}a$ allows to follow an a-edge in the graph in backward direction. In the following definition, we only allow forward navigation. A *directed tree* is a connected, directed graph, such that there is a unique node without incoming edges (the root) and, for all edges (u, v) and (u', v), we have that $u = u'$ (every node has at most one parent).

Definition 3.4.1. A *conjunctive regular path query (CRPQ)* is a SPARQL pattern that only uses triple patterns, the operator And, and property paths.

The *directed canonical graph of a CRPQ* P is the directed graph obtained from the edges $E \cup E_p$, where $E = \{(x, y) \mid (x, \ell, y)$ is a triple pattern in P and $\ell \in \mathcal{I} \cup \mathcal{V}\}$ and $E_p = \{(x, y) \mid (x, pp, y)$ is a property path pattern in $P\}$.

Definition 3.4.2. A CRPQ P is a *tree pattern query* if

- its directed canonical graph is a directed tree and
- every property path is a concatenation of IRIs and property paths of the form a^* where a is an IRI.

Multi-Source Collection Our analysis shows that 99.77% (99.91%) of the CRPQs have a canonical graph that is an *undirected* tree. Out of these,

87.92% (84.96%) are tree pattern queries. This is a fairly significant number, considering that we require the shape to be a *directed* tree. If we additionally allow the Filter operator (in a similar way as in Section 4.2.1), these percentages remain roughly the same.

Wikidata Collection Results for tree patterns in Wikidata are more complex than the results for the Multi-Source Collection due to the nature of the split between organic and robotic queries, which is why Table 3.5 is included.

Organic logs are covered a slight bit less by tree pattern than robotic ones. However, there are more tree patterns for organic queries in Unique logs. Surprisingly the percentages changes by around 10-20% if we go to Unique logs. And while there are more tree patterns for organic queries in Valid logs, in Unique logs robotic queries have more tree patterns. This was unexpected. As a side note: Some testing suggests that the observation from the last paragraph could to shift if our definition of tree patterns would permit the inclusion of the choice operator. Another deeper study in the future may be interesting.

Log	#Organic	%Organic	#Robotic	%Robotic
Valid	63,682	74.48%	58,855,503	70.56%
Unique	30,476	84.33%	20,944,987	90.95%

Table 3.5: Tree patterns in Wikidata logs split by type, showing results for *Valid* and *Unique*.

4

Structural Analysis

After looking at general high-level aspects of queries, the analysis now turns to the inner structure and complexity of queries. To this end, we will now first focus on defining the structures and properties for structures, as well as the complexity of these categorizations. These characterizations allows to divide the data sets into distinct sets with different properties. After these preparations, the analysis is applied. As it turns out, most queries can be characterized by simple shapes that build on each other.

4.1 Query Structure

How do you analyze the structure of a query? There are a multitude of concerns that have to be considered. Queries need to be carefully transformed to a representation that can be analyzed. We will first start out with simple queries that have few parts, and we will gradually add parts.

Starting off, we prepare for an analytical approach to analyze all queries by dividing them into large fragments with a clear understanding of complexity of queries in them, with the goal that these fragments should cover most queries in collection of logs. The idea is that this allows to understand how complex large parts of real logs are, how they are different, and why they are different based on their division. Finally, this allows to isolate a very small fraction of logs that may contain highly unique and interesting queries for further research by looking at the remainder.

SPARQL patterns of queries that are restricted to using only triple patterns and the operators And, Optional, and Filter- and, in particular, not using subqueries or property paths - received considerable attention in the literature (see, e.g., [BPS15]; [KK16]; [KPS16]; [Let+13]; [PAG09]). Such And, Optional, or Filter patterns will be referred to as *And/Optional/Filter patterns*

or, for succinctness, *AOF patterns*. For instance, the Multi-Source Collection has 200,641,891 (64,857,889) AOF patterns, which amounts to 62.31% (74.09%) of the Select, Ask, and Construct queries.

In Section 4.2 and Section 4.2.3 the graph- and hypergraph structure of AOF patterns is investigated. The graph structure gives a clear view on how such queries are structured and can tell how complex such queries are to evaluate. However, for a significant portion of queries, the graph structure is not adequate to capture their complexity (cf. Example 4.1.1), thus their hypergraph structure needs to be examined. Since the graph structure may be easier to understand and is often sufficient, the graph structure is preferred in all analyses if it is applicable.

Some background on the relationship between the (hyper)graph structure of queries and the complexity of their evaluation: Evaluation of CQ queries is NP-complete in general [CM77], but becomes PTIME if their *hypertreewidth* (see Section 4.2.2) is bounded by a constant [GLS02]. The hypertreewidth measures how close the query is to a tree, i.e. the lower the width, the closer the query is to a tree. Several state-of-the-art join evaluation algorithms (e.g., [Abe+16a]; [KEK17]) effectively use the hypergraph structure of queries to improve their performance, even in the context of RDF processing [Abe+16b]. In Section 4.1.4 is used to establish that there are significant performance differences in today's query engines, even when the hypertreewidth of queries just increases from one to two.

4.1.1 Graph and Hypergraph of a Query

Most queries can be accurately represented as a graph, but in some cases they are better expressed as a hypergraph. We recall: An *(undirected) graph* G is a pair (V, E) where V is its (finite) set of nodes and E is its set of edges where an edge e is a set of one or two nodes, i.e., $e \subseteq V$ and $|e| = 1$ or $|e| = 2$. A *hypergraph* $\mathcal{H}$ consists of a (finite) set of nodes $\mathcal{V}$ and a set of hyperedges $\mathcal{E} \subseteq 2^{\mathcal{V}}$, that is, a hyperedge is a set of nodes.

Most SPARQL patterns do not use variables as predicates, that is, they use triple patterns (s, p, o) where p is an IRI. In particular, consider the case $p \in \mathsf{vars}$ if p is not used elsewhere in the query. In this case, p serves as a wildcard, possibly binding to a value that is returned to the output. We call such patterns *graph patterns*. Evaluation of graph patterns is tightly connected to finding an embedding of the graph representation of the query into the data. In particular, it consists of finding an embedding of the directed and edge-labeled variant of the graph, but the directions and labels of edges can be omitted for simplicity, because they do not influence the structure and cyclicity of graph patterns.

The *triple graph* of graph pattern P is defined to be the following graph: $E = \{\{x, y\} \mid (x, \ell, y)$ is a triple pattern in P and $\ell \in \mathcal{I} \cup \mathcal{V}\}$ and $V = \{x \mid \{x, y\} \in E\}$.

Hypergraph representations can be considered for all AOF patterns. The *triple hypergraph* of a SPARQL pattern P is defined as $\mathcal{E} = \{X \mid$ there is a triple pattern t in P, such that X is the set of blank nodes and variables appearing $t\}$ and $\mathcal{V} = \cup_{e \in E} e$.

Canonical Graphs For several types of queries, the structure of their triple graph will be analyzed. However, the usage of some keywords in certain queries (notably, Filter and Values) can put additional constraints on the query that are not reflected in the triple (hyper)graph and therefore will need to be augmented with additional (hyper)edges. We will call the resulting graphs the *canonical (hyper)graphs* of the queries. For CQ queries however, these keywords are not present, therefore we define their canonical (hyper)graph to be equal to their triple (hyper)graph.

Example 4.1.1. Consider the following (synthetic) CQ queries:

```
ASK WHERE {
    ?x1  :a ?x2 . ?x2 :b ?x3 . ?x3  :c ?x4 }

ASK WHERE {
    ?x1 ?x2 ?x3 . ?x3 :a ?x4 . ?x4 ?x2 ?x5 }
```

In Figure 4.1, in the *top left* the canonical graph of the first query is depicted, which is a sequence of three edges. (The edges are labeled with their labels in the query to improve understanding.) The *bottom left* graph in Figure 4.1 depicts the second query, and it shows why variables on the predicate position in triples cause problems, so they are not considered for canonical graphs. The topological structure of this graph as depicted is a sequence of three edges, just as for the first query. But this completely ignores the join condition on $?x2$. For this query, the canonical hypergraph in Figure 4.1 on the right correctly captures the cyclicity of the query.

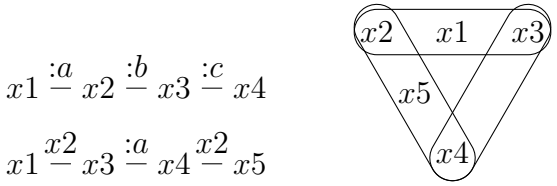


Figure 4.1: Canonical graphs and hypergraph for queries.

4.1.2 Query Fragments

We now discuss the classes of queries for which their canonical graph- and hypergraph structures will be investigated in Section 4.2. The focus will be on fragments of AOF patterns, plus a mild extension for Multi-Source Collection, and another extension for Wikidata Collection. The former only extends to the Values blocks, while the latter also considers Service and Bind in addition to Values. The reasoning for extending the Multi-Source Collection and the Wikidata Collection differently is explained in in Section 2.3.2. In short, the extensions with these operator compositions are covering most of the query logs in their respective collections.

Conjunctive Queries The simplest queries that are considered are the CQ queries, which motivated the classical literature on query evaluation and hypertree structure [CM77]; [GLS02]. For instance, we discovered that 61.00% (60.99%) of the AOF patterns are CQ queries in the Multi-Source Collection.

Definition 4.1.2. A CQ is *suitable for graph analysis* if it is a graph pattern. For a CQ that is suitable for graph analysis, its *canonical graph* is defined as its triple graph. For every other CQ, its *canonical hypergraph* is defined as its triple hypergraph.

Next, the above terminology is extended for CQ queries with Filter, Optional, and Values. Only canonical (hyper)graphs for queries will be considered for which the relationship between efficient query evaluation and their (hyper)graph structure is still similar as for CQ queries. However, this requires some care, especially when considering Optional [BPS15]; [PAG09].

Adding Filter $\mathsf{CQ_F}$ patterns can be evaluated similarly to CQ queries, but the fragment for the analysis of graph shapes still needs to be discussed. A filter constraint R is *simple* if vars(R) contains at most two variables. An almost identical class of queries was also considered in [PV11].

Definition 4.1.3. A $\mathsf{CQ_F}$ query is *suitable for graph analysis* if it is a graph pattern and all filter constraints are simple. For such a $\mathsf{CQ_F}$ query, the *canonical graph* is its triple graph, with an additional edge $\{x, y\}$ for each filter constraint that uses two variables x and y. For all other $\mathsf{CQ_F}$ queries, its *canonical hypergraph* is obtained from its triple graph, with an additional hyperedge $\{x_1, \ldots, x_k\}$ for each filter constraint that uses precisely the k variables $x_1, \ldots, x_k$.

In the Multi-Source Collection, 81.07% (90.28%) of the $\mathsf{CQ_F}$ patterns are suitable for graph analysis.

Adding Optional We now additionally consider Optional. Pérez et al. [PAG09] showed that unrestricted use of Optional in SPARQL patterns makes query evaluation PSPACE-complete, which is significantly more complex than the NP-completeness of CQ or $\mathsf{CQ_F}$ queries. They discovered that patterns that satisfy an extra condition called *well-designedness* [PAG09], can be evaluated more efficiently. Letelier et al. [Let+13] show that, in the presence of projection, evaluation of well-designed patterns is Σ_2^P-complete [Let+13].

Definition 4.1.4. A SPARQL pattern P using only the operators And, Filter, and Optional is *well-designed* if for every occurrence i of an Optional-pattern (P_1 Optional P_2) in P, the variables from $\mathsf{vars}(P_2) \setminus \mathsf{vars}(P_1)$ occur in P only inside i.[a]

[a] Perez et al.'s definition also has a safety condition on the filter statements of the patterns, but the omission of this condition does not affect the results.

Example 4.1.5. Consider the pattern

$$((?x, b, ?y)\ OPT\ (?y, c, ?z))\ OPT\ (?x, d, ?z)$$

Then $?z$ from $(?y, c, ?z)$ is not in $(?x, b, ?y)$, but it is outside in $(?x, b, ?z)$. Therefore, the pattern is not well-designed.

In the Multi-Source Collection 98.74% (98.18%) of the AOF patterns are well-designed (but do not necessarily have simple filters).

However, in the Wikidata Collection, there are substantial amounts of queries that go beyond only using the operators And, Optional, and Filter. In the Wikidata Collection, only 27.72% (44.24%) of the queries are in the pure $\mathsf{CQ_F}$ fragment. We will see how the Wikidata Collection can be handled to cover more queries later in this section. For now, we will continue to look at the AOF patterns.

Unfortunately, it is not yet sufficient for well-designed patterns to have a hypergraph of constant hypertreewidth for their evaluation to be tractable [BPS15]. However, Barceló et al. show that this can be mended by an additional restriction called *bounded interface width*. This notion will be explained by an example, for details refer to [BPS15].

Example 4.1.6. The following patterns come from [Let+13]; [PAG09]:

$$P_1 = ((?A, name, ?N)\ OPT\ (?A, email, ?E))\ OPT\ (?A, webPage, ?W)$$

$$P_2 = (?A, name, ?N)\ OPT\ ((?A, email, ?E)\ OPT\ (?A, webPage, ?W))$$

Figure 4.2 has tree representations T_1 and T_2 for P_1 and P_2, respectively, called *pattern trees*. The pattern trees T_i are obtained from the parse trees of P_i by applying a standard encoding based on Currying [MN07, Section 4.1.1]. The encoding only affects the arguments of the Optional operators in the queries. If the query also uses And, then it should first be brought in *Optional-normal form* [PAG09] and then turned into a pattern tree. The resulting pattern trees will then have a CQ in each of its nodes.

Barceló et al. define pattern trees to be *well-designed* if, for each variable, the set of nodes in which it occurs forms a connected set. Notice that this is the case for T_1 and T_2. It would be violated in T_1 if the root would not use the variable ?A. Likewise, it would be violated in T_2 if the node labeled (?A, email, ?E) would not use the variable ?A.

The *interface width* of the pattern trees is the maximum number of common variables between a node and its child. Both trees in Figure 4.2 (and both queries P_1 and P_2) therefore have interface width one. (Common variables are bold in Figure 4.2.) If T_1 would use variable $?W$ instead of $?N$, then its interface width would be two.

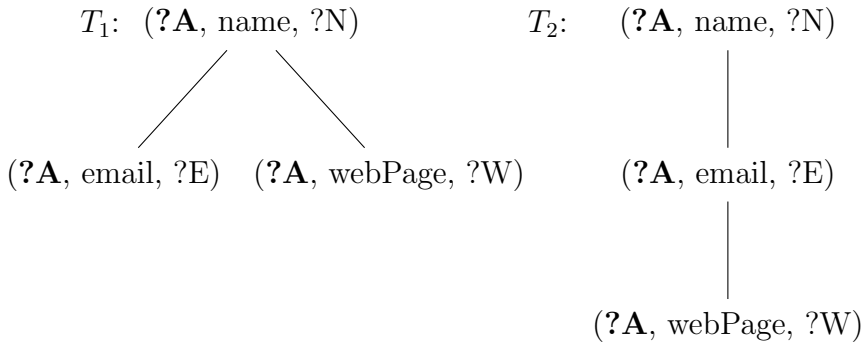


Figure 4.2: Pattern trees that correspond to the queries.

Definition 4.1.7. A SPARQL pattern P using only the operators And, Filter, and Optional is a CQ_{OF} *query* if it has a well-designed pattern tree with interface width 1. It is *suitable for graph analysis* if it is a graph pattern and all its filter conditions are simple. The canonical graph and hypergraph of a $\mathsf{CQ_{OF}}$ query is defined analogously to that of $\mathsf{CQ_F}$ queries. That is, its triple graph (resp., hypergraph) is augmented with edges $\{x, y\}$ (resp., hyperedges $\{x_1, \ldots, x_k\}$) for each filter constraint that uses precisely the variables x and y (resp., $x_1, \ldots, x_k$) for each filter constraint.

For the Multi-Source Collection, it turns out that 98.72% (98.13%) of the AOF patterns are $\mathsf{CQ_{OF}}$ queries, which is almost equal to the number of well-designed patterns. Moreover, 85.30% (93.87%) are $\mathsf{CQ_{OF}}$ patterns that are suitable for graph analysis.

Adding Values The Values keyword was used in 2.24% (5.61%) of the queries in the Multi-Source Collection. It is used particularly often in DBpedia17, where it appears in 4.03% (13.37%) of the queries. The usage of Values in Wikidata Collection is very similar, it is used 4.56% (4.58%).

The purpose of Values blocks is to test if a variable (or a tuple of variables) appears in a set that is given in the query. For instance, the Values block

```
VALUES (?country) {
    "Belgium" "France" "Germany"
}
```

restricts the variable `?country` to be assigned to one of the values `"Belgium"`, `"France"`, or `"Germany"`. Values blocks are used almost exclusively for unary conditions, that is, to test if the value of a single variable is in a given set of constants. However, it can also be used to test higher arity constraints, as in the Values block

```
VALUES (?x ?y) {
  (:a :b)
  (:a :c)
}
```

which imposes a *binary* constraint, i.e., it binds the variable pair `(?x ?y)` to one of the two pairs in the body of the Values block. Concerning the shape analysis, Values blocks that use constraints of arity two or less are distinguished from the others.

Definition 4.1.8. A *CQ_{OFV} query* is a SPARQL pattern P using only the operators And, Filter, Optional, and Values, such that the pattern obtained from P by removing all Values blocks is a $\mathsf{CQ_{OF}}$ query. It is *suitable for graph analysis* if all filters are simple and all values blocks have arity at most two. If a $\mathsf{CQ_{OFV}}$ query is suitable for graph analysis, its *canonical graph* is obtained from the triple graph by augmenting it with an edge for each binary filter constraint, and an edge for each binary Values block. For every other $\mathsf{CQ_{OFV}}$ queries, its *canonical hypergraph* is obtained from the triple hypergraph by augmenting it with a hyperedge $\{x_1, \ldots, x_k\}$ for each filter- or values block that uses precisely the variables $x_1, \ldots, x_k$.

Adding Service For the Multi-Source Collection, Service was used very rarely. Therefore it was not considered when analyzing the logs. However, this changes when going to the Wikidata Collection. Service is used extensively in Wikidata queries, most commonly for Wikidata's labeling service. For this reason, Bielefeldt et al. [BGK18] ignore the labeling service entirely in their co-occurrence analysis of SPARQL features. Similarly to before, a Service element S is k-ary if it contains k variables. All unary or binary Service elements are suitable for graph analysis. When considering the graph of patterns with Service, edges of the form $\{x, y\}$ for all binary Service elements are added, in which x and y are the variables. Higher arity Values conditions are considered in Section 4.2.2. We will revisit this operator soon, when we take a closer look at what queries are dominant in the Wikidata Collection.

Adding Bind For the Multi-Source Collection, Bind was not considered, but is considered for the Wikidata Collection. Concerning Bind, the approach is similar to Filter. A Bind-condition is k-ary if it involves k variables. Unary and binary Bind-conditions are considered to be suitable for graph analysis and are materialized as edges in the graph. Higher-arity Bind-conditions are considered in Section 4.2.2.

Definition 4.1.9. A *CQ_{OF+} query* is an extension of a $\mathsf{CQ_{OFV}}$ query. It also permits Bind and Service similarly to Filter and Values: If all edges induced by Bind and Service are at most binary, the query is suitable for graph analysis and the edges are added to the canonical graph of the query. The canonical hypergraph is obtained by adding hyperedges induced by by Bind and Service to the hypergraph.

Operators in Wikidata Collection If you look at the Wikidata Collection, even after adding Filter and Optional, only 53,804,198 (14,649,616) queries are eligible for graph shape analysis. This is still only 25.87% (42.32%) of the data set. This is because of the high usage of property paths. We will now see how property paths (recursion), Bind, Service, and Values can be incorporated to increase the number of suitable queries to 177,022,071 (30,540,864), or 85.11% (88.22%) of these logs.

Property Paths Property paths are unproblematic. Conjunctive queries extended with property paths closely correspond to the well known *conjunctive two-way regular path queries (C2RPQs)*, which form a basis of navigational query languages for graphs. Indeed, property paths are very closely related to regular path queries and, due to the ^-operator, they can navigate edges in both forward and backward direction, which makes them *two-way*.

Although there are some semantical differences between two-way regular path queries and property paths [ACP12]; [HS13]; [LM13], these differences are not crucial for the analysis.

Definition 4.1.10. A *conjunctive two-way regular path query (C2RPQ)* is a SPARQL pattern that only uses triple patterns, the operator And, and property paths.

Every C2RPQ that is a graph pattern is suitable for analysis. The *graph of a C2RPQ* P is obtained from G_P by adding the edges $\{\{x, y\} \mid (x, pp, y)$ is a property path pattern in $P\}$ to E_P (and adding nodes to V_P if necessary).

Definition 4.1.11. By C2RPQ+ we denote the entire fragment that uses And, Optional, Filter, *property paths*, Bind, Service, and Values and that is suitable for graph analysis.

If you look at the Wikidata Collection, in total, this amounts to 176,679,495 (30,371,003) robotic and 342,576 (169,861) organic queries, which make up 85.22% (88.39%) and 51.50% (65.22%) of the robotic and organic queries, respectively.

The fragments $\mathsf{CQ_F}$, $\mathsf{CQ_{OF}}$, and $\mathsf{CQ_{OF+}}$ can be naturally extended like going from CQ to C2RPQ, by permitting property paths. These extensions are called $\mathsf{C2RPQ_F}$, $\mathsf{C2RPQ_{OF}}$, and C2RPQ+.

The Set of Shape Analysis Experiments For the Wikidata Collection the graphs of 8 fragments were analyzed, namely CQ, $\mathsf{CQ_F}$, $\mathsf{CQ_{OF}}$, $\mathsf{CQ_{OF+}}$,

C2RPQ, C2RPQ$_{\mathsf{F}}$, C2RPQ$_{\mathsf{OF}}$, C2RPQ+. All fragments were analyzed across three dimensions: (1) *Robotic* versus *Organic*, (2) Valid versus Unique, and (3) *with constants* versus *without constants*. In the analyses without constants, all nodes in the graphs that originated from IRIs or literals were removed. Furthermore, for all fragments, the time-out (TO) queries were analyzed separately from the successful (OK) queries. This results in 64 separate runs of the shape analysis (8 fragments with three splits for three different dimensions).

Finally, to finish this section, Figure 4.3 illustrates how the query fragments are related.

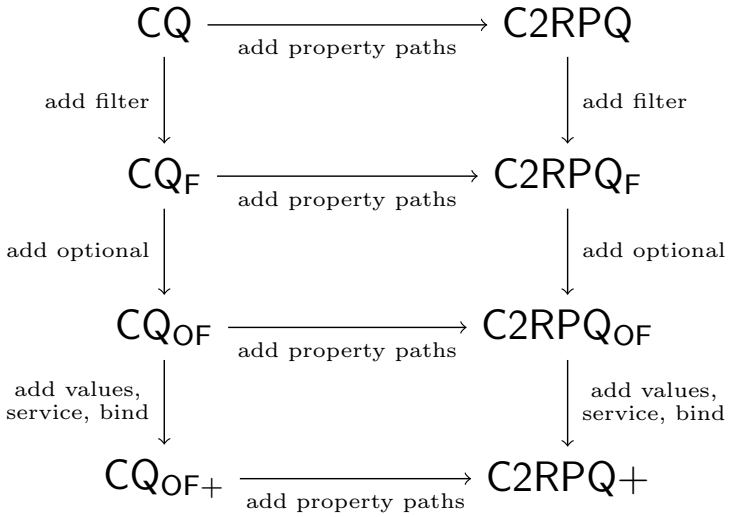


Figure 4.3: Query fragments for shape analysis. Inclusions are represented by arrows.

4.1.3 Well-Designedness and Unions

In this section we will take a look at the usage of well-designedness of the queries in the logs. Kaminski and Kostylev [KK16] defined a weaker version of well-designedness that has similar favorable computational properties. Therefore it was also analyzed whether queries are *weakly well-designed*.

The definition for weakly well-designed patterns is similar to Definition 4.1.4, but it introduces the notion of pattern domination. Given a pattern P, an occurrence i_1 in P dominates another occurrence i_2 if there exists an occurrence j of an OPT-pattern, such that i_1 is inside the left argument of j, and i_2 is inside the right argument. A pattern P is *weakly well-designed* if the outside occurrences i only appear in

- subpatterns whose occurrences are dominated by i, and

- constraints of top-level occurrences of FILTER-patterns.

If you apply this relaxed definition to Example 4.1.5, then the pattern is weakly-well designed, because $(?y, c, ?z)$ dominates over $(?x, b, ?y)$.

Multi-Source Collection Table 4.1 shows the number of AOF queries (and the percentages thereof) that are well-designed (*wd*) and weakly well-designed (*wwd*). We also took the set of queries that only use And, Optional, Filter, and Union (AOFU in Table 4.1) and investigated the percentages of queries thereof that are *unions* of wd or wwd queries. In most cases where the query is not a union of wd or wwd queries, it is because the union is not the top-level operator.

Property	#Valid	%Valid	#Unique	%Unique
wd	198,109,323	98.74%	63,677,171	98.18%
wwd	200,064,814	99.71%	64,749,468	99.83%
AOF	200,641,891	100.00%	64,857,889	100.00%
u*wd*	208,672,931	74.35%	69,279,286	88.72%
u*wwd*	210,638,343	75.05%	70,360,134	90.10%
AOFU	280,672,732	100.00%	78,088,794	100.00%

Table 4.1: Well-designedness (*wd*), weak well-designedness (*wwd*), and unions thereof (Multi-Source Collection).

Wikidata Collection Table 4.2 summarizes the well-designedness of queries in the Wikidata Collection.

An interesting observation is that the weakly well-designed queries in robotic queries cover practically 100% in the logs. In general, the results are quite similar to the results for the Multi-Source Collection in Table 4.1. However, the results for unions of well-designed queries appear to be higher on first glance, but this is due do the fact that AOFU queries are not much higher than AOF queries. Although Union appears in almost 10% of the queries (see Table 2.4), it does not occur in combination the operators And, Filter, and Optional, which can be seen in Table 2.7 for combinations with And and Filter, And the combination of And, Filter, and Optional together. Note that Table 2.7 only provides a subset of combinations, meaning that combinations with other operators are not included.

Property	#O V	%O V	#R V	%R V
wd	122,519	96.90%	54,177,960	94.17%
wwd	126,234	99.84%	57,527,655	100.00%
AOF	126,436	100.00%	57,530,280	100.00%
u*wd*	123,427	94.36%	54,195,498	93.24%
u*wwd*	127,142	97.20%	57,545,193	99.00%
AOFU	130,811	100.00%	58,125,339	100.00%

Property	#O U	%O U	#R U	%R U
wd	56,729	96.42%	15,030,107	98.51%
wwd	58,708	99.78%	15,256,298	99.99%
AOF	58,838	100.00%	15,257,255	100.00%
u*wd*	57,128	94.31%	15,036,064	98.07%
u*wwd*	59,107	97.58%	15,262,255	99.54%
AOFU	60,573	100.00%	15,332,293	100.00%

Table 4.2: Well-designedness (*wd*), weak well-designedness (*wwd*), and unions thereof (***u****wd* and ***u****wwd*) (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

4.1.4 Benchmarking: Impact of Query Structure

In this section, we study the impact of query structure on actual evaluation in real-world systems.

A set of experiments were conducted, aiming at comparing the execution times of conjunctive queries whose canonical graphs exhibit specific shapes. Because of their simplicity and prevalence, chain and cycle queries were chosen for this empirical study.

A *chain query (of length k)* is a CQ for which the canonical graph is isomorphic to the undirected graph with edges $\{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{k-1}, x_k\}$. A *cycle query (of length k)* is a CQ for which the canonical graph is isomorphic to $\{x_0, x_1\}, \ldots, \{x_{k-1}, x_0\}$. As an edge case, chains of length zero are also allowed. Such chains consist either of a single node or no node at all.

These shapes have been selected as representatives of the queries with hypertreewidth 1 and 2, respectively, and have also been used to compare the performances of join algorithms in other studies, e.g., [KEK17].

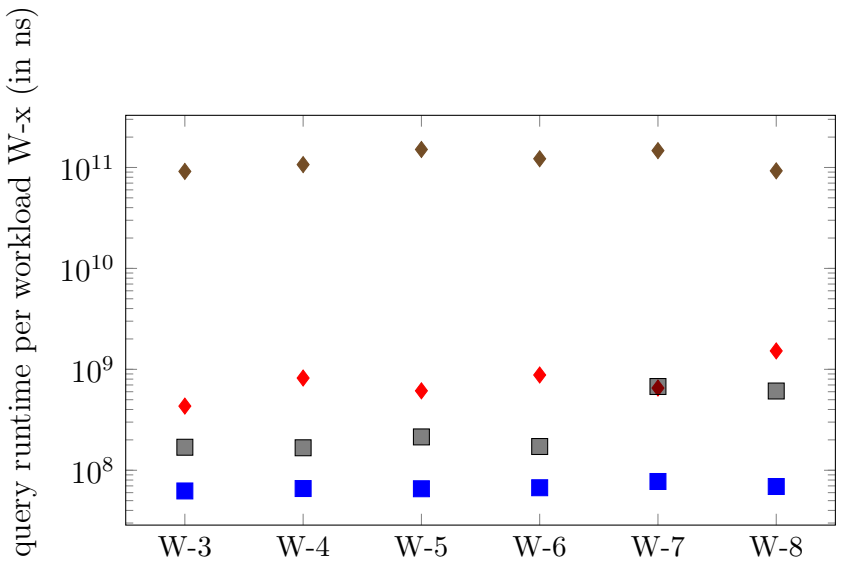
Experimental Setup In order to generate query workloads containing the aforementioned types of queries, gMark [Bag+17] was used, a publicly available[1] schema-driven generator for graph instances and graph queries. gMark was tuned to generate diverse query workloads, each containing 100 chain and cycle queries, respectively.[2] Each workload has been generated by using chains and cycles of different length varying from 3 to 8.

In these experiments, two opposite graph database systems were considered and contrasted, namely PostgreSQL [Pos18], an open-source relational DBMS, and Blazegraph [Sys17], an high-performance SPARQL query engine powering the Wikimedia's official query service [VK14] and thus used for Wikidata real-world queries. These experiments were run on a dual-CPU Intel Xeon E5-2630v2 2.6 GHz server[3] with 128GB RAM and running Ubuntu 16.04 LTS. For the experimental setup, the version of PostgreSQL was 9.3 and the version of Blazegraph was 2.1.4. In the gMark configuration [Bag+17] the use base *Bib* was employed for the schema of the generated graph (of size $100k$ nodes) and for the generated queries as well. The query workloads in SQL and SPARQL was employed as generated by gMark after the elimination of empty unions (since gMark is geared towards generating UCRPQs) and of the keyword Distinct in the body of the queries. For the experimental setup, it seemed to be a good idea to focus one query type at a time to keep results compatible for comparison, but since gMark normally generates a mixed workloads of Select/Ask queries, the Select clauses were transformed to compatible Ask clauses.

[1] `https://github.com/graphMark/gmark`

[2] Recall that gMark can generate queries of four shapes: chain, star, chain-star and cycle. Thus chain queries were cherry-picked as representatives of queries with hypertreewidth equal to 1.

[3] Every CPU has 6 physical cores and, with hyperthreading, 12 logical cores.



W-x	W-3	W-4	W-5	W-6	W-7	W-8
%t/o	18%	34%	43%	39%	43%	30%

Figure 4.4: Execution times (top) of diverse workload of chain/cycle queries (of length 3,4,5,6) on Blazegraph (BG) and PostgreSQL(PG). Number of timeouts per workload for CyclePG only (bottom). CyclePG times include t/o of 300s (per query).

Experimental Results Figure 4.4 (top) depicts the average runtime (in ns, log scale) of the workloads of chain (cycle, resp.) queries with length from 3 to 8 on Blazegraph (BG) and PostgreSQL (PG). It can be observed that the overall performance of Blazegraph is superior to that of PostgreSQL. Indeed, in PostgreSQL many cycles queries are timed out (after 300s per query), and one can expect that the real overall performance of PostgreSQL is even worse than the results reported in Figure 4.4. Figure 4.4 (bottom) reports the percentages of the number of reached timeouts for workloads of cycle queries of various sizes when executed in PostgreSQL.

It is worthwhile observing that for both systems the difference between average runtime of chain query workloads and cycle query workloads is non negligible, thus confirming that the graph representation and the shape of queries cannot be ignored. This experiment also motivated to dig deeper in the shape analysis of the query logs, which is reported in Section 4.2.

4.2 Shape Analysis

In this section, the results of the analysis of the shapes of the canonical graphs and the tree- and hypertreewidth are presented. For Multi-Source Collection, this is done on CQ, $\mathsf{CQ_F}$, $\mathsf{CQ_{OF}}$, and $\mathsf{CQ_{OFV}}$ queries, while on the Wikidata Collection, the $\mathsf{CQ_{OFV}}$ fragment is extended to $\mathsf{CQ_{OF+}}$, and the analysis is also done on C2RPQ, $\mathsf{C2RPQ_F}$, $\mathsf{C2RPQ_{OF}}$, and C2RPQ+ queries,

To start of, a note on the size of queries in the Multi-Source Collection. Figure 4.5 shows the respective sizes of these queries that have at least two triples by considering both *Valid* and *Unique* queries side by side. The fractions of queries with one triple are 90.65% (85.36%), 87.71% (83.22%), 81.54% (76.99%) and 81.81%(77.81%) for CQ, $\mathsf{CQ_F}$, $\mathsf{CQ_{OF}}$ and $\mathsf{CQ_{OFV}}$ respectively. Unsurprisingly, small queries are more likely to be in one of these fragments and, therefore, simple queries are represented even more in these data sets than in the overall data set. Nevertheless, there are CQ and $\mathsf{CQ_F}$ queries with up to 81 triples and $\mathsf{CQ_{OF}}$ and $\mathsf{CQ_{OFV}}$ queries with up to 211 triples. This is much larger than what was observed for the Wikidata Collection in Section 2.3.3.

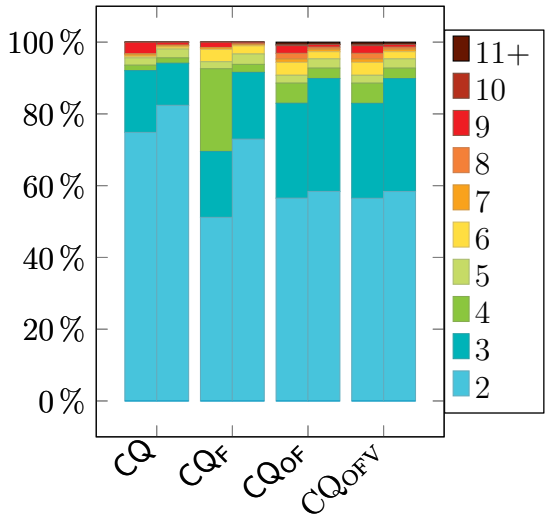


Figure 4.5: Size of *Valid* (versus *Unique*) CQ-like queries with at least two triples.

4.2.1 Query Graph Shape Classification

We first recall or define the basic shapes of the canonical graphs that we will study in this section. The shapes *chains* and *cycle* are already defined in

Section 4.1.4. A *chain set* is a graph in which every connected component is a chain (so, each chain is also a chain set).

A *tree* is an undirected graph, such that, for every pair of nodes x and y, there exists exactly one undirected path from x to y (hence, every chain is also a tree). A *forest* is a graph in which every connected component is a tree. A *star* is a tree for which there exists at most one node with more than two neighbors, that is, there is at most one node u, such that there exist u_1, u_2, and u_3, all pairwise different and different from u, for which $\{u, u_i\} \in E$ for each $i = 1, 2, 3$.

Inspired by the results obtained with gMark on synthetic queries in Section 4.1.4, the analysis of the query logs proceeded by looking at the encountered query shapes. Here, we consider queries as edge-labeled graphs, as defined in Section 4.1.1. The hypergraph structure will also be analyzed following soon in Section 4.2.2.

Multi-Source Collection In this section the graph structure of queries is analyzed. For this, only graphs for queries are considered that were defined to be *suitable for graph analysis* in Section 4.1.1. The remaining 6.96 million queries in $\mathsf{CQ_{OF}}$ are considered in Section 4.2.2.

CQ queries, $\mathsf{CQ_F}$ queries, $\mathsf{CQ_{OF}}$ queries, and $\mathsf{CQ_{OFV}}$ queries are investigated. The last three fragments are interesting, because they bring more queries under scrutiny than the plain CQ set of query logs (by an increase of roughly 33% (50%), 44% (64%), and 47% (70%) respectively). The first candidates for classification for identification are classical query shapes, such as all variants of tree-like shapes (single edges, chains, sets of chains, stars, trees, and forests). The results are summarized in Table 4.3 and Table 4.4. From the results, the following observations can be drawn: While tree-shaped queries even in their simple forms (chain of length 1 or single edges) are very frequent, the only observed exception occurs with star queries, which have very low occurrence with respect to the other tree-like shapes.

Since simple queries are overrepresented in query logs (already over 87.76% (83.23%) of $\mathsf{CQ_F}$ patterns use only one triple, for example), it is no surprise that the overwhelming majority of the queries is acyclic, i.e., a forest. However, to also get a better understanding of the more *complex* and therefore potentially interesting queries in the logs, further investigation has to be done on the cyclic queries. The goal is to obtain a cumulative shape analysis where simpler shapes are subsumed by more sophisticated query shapes, with the latter reaching almost 100% coverage of the query logs.

A first observation was that plain cycles are not very common. By visually inspecting the remaining cyclic queries, it was observed that many of them

VALID	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	73,147	0.06%	73,155	0.05%	73,155	0.04%	74,891	0.04%
≤ 1 edge	107,268,916	90.71%	137,634,760	87.56%	139,234,499	81.35%	141,642,411	81.49%
chain	116,816,836	98.78%	151,963,617	96.68%	159,787,714	93.36%	162,216,710	93.32%
star	117,683,253	99.52%	155,325,069	98.82%	168,220,691	98.29%	170,671,088	98.19%
tree	118,059,399	99.83%	155,716,314	99.07%	168,936,241	98.71%	171,386,859	98.60%
flower	118,225,680	99.98%	156,730,621	99.71%	170,174,607	99.43%	172,922,659	99.48%
chain set	116,835,460	98.80%	151,990,203	96.70%	159,931,312	93.45%	162,287,197	93.36%
forest	118,078,726	99.85%	155,748,689	99.09%	169,089,411	98.80%	171,466,918	98.64%
bouquet	118,245,059	99.99%	156,763,406	99.73%	170,328,206	99.52%	173,003,151	99.53%
tw ≤ 2	118,254,672	100.00%	157,183,767	100.00%	171,147,726	100.00%	173,822,690	100.00%
tw ≤ 3	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%
total	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%

Table 4.3: Cumulative shape analysis of graph patterns in CQ, CQ_F, CQ_{OF}, and CQ_{OFV} across all logs of Multi-Source Collection (*Valid*). The relative numbers are w.r.t. the queries that are suitable for graph analysis.

UNIQUE	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	1,279	0.00%	1,284	0.00%	1,284	0.00%	1,661	0.00%
≤ 1 edge	31,785,575	85.41%	46,480,574	83.05%	46,772,128	76.82%	48,866,909	77.37%
chain	36,839,344	98.99%	54,131,560	96.72%	56,042,768	92.05%	58,142,029	92.06%
star	37,123,785	99.75%	55,417,051	99.02%	60,203,786	98.89%	62,306,677	98.65%
tree	37,184,810	99.92%	55,487,815	99.15%	60,311,400	99.06%	62,414,439	98.82%
flower	37,202,015	99.96%	55,892,860	99.87%	60,735,713	99.76%	63,010,697	99.77%
chain set	36,851,176	99.02%	54,150,770	96.76%	56,096,837	92.14%	58,196,121	92.15%
forest	37,197,115	99.95%	55,509,443	99.19%	60,370,204	99.16%	62,473,266	98.92%
bouquet	37,214,357	100.00%	55,914,792	99.91%	60,794,835	99.86%	63,069,846	99.86%
tw ≤ 2	37,216,150	100.00%	55,965,143	100.00%	60,881,508	100.00%	63,156,533	100.00%
tw ≤ 3	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%
total	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%

Table 4.4: Cumulative shape analysis of graph patterns in CQ, CQ_F, CQ_{OF}, and CQ_{OFV} across logs of Multi-Source Collection (*Unique*). The relative numbers are w.r.t. the queries that are suitable for graph analysis.

VALID	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	106,952,766	90.44%	136,357,792	86.75%	144,549,634	84.46%	144,643,932	83.21%
≤ 1 edge	116,643,820	98.64%	150,954,951	96.04%	160,737,562	93.92%	163,386,730	94.00%
chain	117,774,655	99.59%	155,832,073	99.14%	167,819,465	98.06%	170,472,931	98.07%
star	117,876,831	99.68%	156,787,151	99.75%	170,062,935	99.37%	172,737,353	99.38%
tree	118,235,060	99.98%	157,146,906	99.98%	170,423,705	99.58%	173,098,147	99.58%
flower	118,243,330	99.99%	157,162,189	99.99%	170,439,245	99.59%	173,114,179	99.59%
chain set	117,785,058	99.60%	155,852,116	99.15%	167,851,157	98.07%	170,504,640	98.09%
forest	118,245,559	99.99%	157,167,354	99.99%	170,732,618	99.76%	173,407,077	99.76%
bouquet	118,253,840	100.00%	157,182,660	100.00%	170,748,181	99.77%	173,423,132	99.77%
tw ≤ 2	118,254,674	100.00%	157,183,769	100.00%	171,147,728	100.00%	173,822,692	100.00%
tw ≤ 3	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%
total	118,254,676	100.00%	157,183,771	100.00%	171,147,730	100.00%	173,822,694	100.00%

Table 4.5: Cumulative shape analysis of graph patterns in CQ, CQ_F, CQ_{OF}, and CQ_{OFV}, *after removal of IRIs*, across all logs in Multi-Source Collection (*Valid*). The relative numbers are w.r.t. the queries that are suitable for graph analysis.

UNIQUE	CQ/graph		CQ_F/graph		CQ_{OF}/graph		CQ_{OFV}/graph	
Shape	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*	*#Queries*	*Relative %*
no edge	32,886,654	88.37%	47,048,004	84.07%	49,453,297	81.23%	49,490,285	78.36%
≤ 1 edge	36,511,703	98.11%	52,939,383	94.59%	56,293,258	92.46%	58,562,031	92.73%
chain	37,150,107	99.82%	55,596,772	99.34%	60,134,203	98.77%	62,405,465	98.81%
star	37,164,017	99.86%	55,898,107	99.88%	60,743,607	99.77%	63,018,260	99.78%
tree	37,210,340	99.98%	55,944,891	99.96%	60,791,019	99.85%	63,065,687	99.86%
flower	37,213,881	99.99%	55,954,879	99.98%	60,801,134	99.87%	63,076,131	99.87%
chain set	37,151,726	99.83%	55,605,967	99.36%	60,148,326	98.80%	62,419,603	98.83%
forest	37,212,024	99.99%	55,954,365	99.98%	60,834,660	99.92%	63,109,343	99.93%
bouquet	37,215,574	100.00%	55,964,373	100.00%	60,844,795	99.94%	63,119,807	99.94%
tw ≤ 2	37,216,152	100.00%	55,965,145	100.00%	60,881,510	100.00%	63,156,535	100.00%
tw ≤ 3	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%
total	37,216,153	100.00%	55,965,146	100.00%	60,881,511	100.00%	63,156,536	100.00%

Table 4.6: Cumulative shape analysis of graph patterns in CQ, CQ_F, CQ_{OF}, and CQ_{OFV}, *after removal of IRIs*, across all logs in Multi-Source Collection (*Unique*). The relative numbers are w.r.t. the queries that are suitable for graph analysis.

could be seen as a node with simple attachments. The query shape looked like a *flower*, so this name was chosen for this shape.

Definition 4.2.1. A *petal* is a graph consisting of a source node s, target node t, and a set of at least two node-disjoint paths from s to t. (For instance, a cycle is a petal that uses two paths.)

A *flower* is a graph consisting of a node x with three types of attachments: chains (the *stamens*), trees that are not chains (the *stems*), and *petals*.

Flowers have a nice property: Their treewidth is limited to two. The cyclic attachments are series-parallel graphs, which have a treewidth of at most two [BLS99]. For a decomposition, you can use the central node and attach the decompositions of the attachments. An example of a real flower query from the DBpedia logs is illustrated in Figure 4.6. It consists of a central node with four petals (one with three paths), ten stamens and zero stems attached.

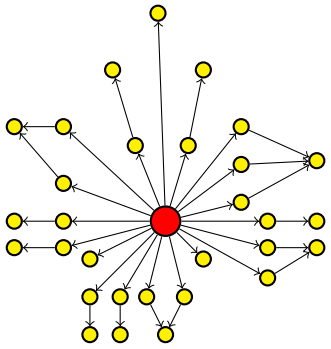


Figure 4.6: An example of a flower query found in DBpedia query logs (arrows were added to indicate the edge directions in the query; labels are omitted).

Like other shapes, sets of flowers were also considered, since set shapes are used for non-connected graphs for the cumulative shape analysis. This set of flowers is simply called *bouquets*. It allows to further increase the ratio of queries that could be classified in the query logs. The number of flowers and bouquets in the query logs only overcome those of trees and forests by roughly 0.01%–0.09% (0.03–0.10%) for all four fragments. However, for all fragments, the majority of the cyclic queries is captured by bouquets.

In the preceding analysis, the shapes of queries were analyzed for graphs of queries as defined in Section 4.1.1, i.e., the nodes can be either variables or constants. Constants are in fact helpful to obtain a rough idea of the shape of patterns that users try to find, but research on query optimization often

focuses on the shape of patterns *without constants*. The reason is that constants can typically be matched to only one node in the graph and therefore do not highly contribute to the complexity of evaluation. For that reason, a second parallel analysis is run on graphs of queries excluding constants in order to identify the differences in the obtained shape classification. Table 4.5 and Table 4.6 show the results for *Valid* and *Unique* logs respectively.

The most significant observation here is that many shapes disintegrate to a set of variables (i.e., no more edges are present in their graph). More precisely, for the four fragments CQ, $\mathsf{CQ_F}$, $\mathsf{CQ_{OF}}$, and $\mathsf{CQ_{OFV}}$, we have that respectively 90.44% (88.37%), 86.75% (84.07%), 84.46 (81.23%), and 83.21% (78.36%) of the queries that are suitable for graph analysis have no more edges when considering the restriction of their canonical graphs to variables only. This is a huge change, since such shapes only constituted 0.00%–0.06% of the shapes of queries with constants in Table 4.3.

Wikidata Collection In the following, we will discuss the results on the valid queries in the largest fragment of the Wikidata Collection, the C2RPQ+ queries. Furthermore, we give some insights about how the results change for the other fragments. The results for the valid C2RPQ+ queries are in Table 4.7. Note that some of the queries were empty (0.8% with constants and 2.44% after removing constants); we did not include them.

In the table, we see several trends that we also observed in the analysis for the other fragments. First of all, in the shapes that include constants, *stars are quite common*. In the C2RPQ+ queries, 85.17% (87.13%) of the organic and 98.02% (99.30%) of the robotic queries are stars. The number of acyclic queries is even larger: consistently over 99% when constants are absent. As opposed to valid queries, in the timeout logs, the number of cyclic queries significantly increases. For organic CQs, the number of cyclic queries goes up to about 10%. This number decreases for more complex fragments, but is still about 7.5% for the unique C2RPQ+ queries (both organic and robotic) and around 3%–4% if constants are removed. Together with the observation from Section 2.3 that valid queries contain 2.58 triples on average, whereas valid timeout queries have 5.65 triples on average, this suggests that cyclicity and query size play an important role in efficient query evaluation.

The logs strongly confirm a hypothesis that is often stated in theoretical research: the cyclic queries in practical applications are only *mildly cyclic*, i.e., with a treewidth less than or equal to k for small values of k. This means that database queries typically do not have large k-cliques encoded in their shape, but remain tree-like. Indeed, the largest treewidth we found in the entire logs was four, for which we found 15 (5) queries.

C2RPQ+ with constants / Valid				
Shape	*#Queries (Organic)*	*Relative % (Organic)*	*#Queries (Robotic)*	*Relative % (Robotic)*
node[0]	0	0.00%	0	0.00%
chain[1]	107,436	31.48%	125,277,683	71.49%
chain	207,292	60.74%	158,150,895	90.25%
star	290,665	85.17%	171,767,410	98.02%
tree	329,701	96.61%	171,885,247	98.09%
flower	335,271	98.24%	172,070,177	98.19%
chain set	209,540	61.40%	158,723,217	90.57%
forest	333,725	97.79%	172,461,994	98.41%
bouquet	339,440	99.46%	172,646,921	98.52%
tw ≤ 2	341,221	99.98%	175,240,211	100.00%
tw ≤ 3	341,268	100.00%	175,240,228	100.00%
tw ≤ 4	341,274	100.00%	175,240,237	100.00%
total	341,274	100.00%	175,240,237	100.00%

C2RPQ+ without constants / Valid				
Shape	*#Queries (Organic)*	*Relative % (Organic)*	*#Queries (Robotic)*	*Relative % (Robotic)*
node[0]	85,601	25.13%	63,048,127	36.58%
chain[1]	200,597	58.89%	155,126,595	89.99%
chain	259,074	76.06%	163,055,757	94.59%
star	312,128	91.63%	170,205,746	98.74%
tree	321,914	94.50%	170,243,847	98.76%
flower	323,830	95.07%	170,413,306	98.86%
chain set	273,992	80.44%	165,014,582	95.73%
forest	337,730	99.15%	172,203,761	99.90%
bouquet	339,661	99.71%	172,373,220	100.00%
tw ≤ 2	340,632	100.00%	172,375,717	100.00%
tw ≤ 3	340,632	100.00%	172,375,717	100.00%
tw ≤ 4	340,632	100.00%	172,375,717	100.00%
total	340,632	100.00%	172,375,717	100.00%

Table 4.7: Cumulative shape analysis of graph patterns in C2RPQ+ across the valid logs in Wikidata Collection.
[0]: without edges; [1]: length ≤ 1

4.2.2 Tree- and Hypertreewidth

It is well-known that the tree- or hypertreewidth of queries are important indicators to gauge the complexity of their evaluation.

For the original definitions and more information about tree- or hypertreewidth, refer to an excellent introduction [GGS14]. We will only look at the definitions briefly in the following.

Definition 4.2.2. A *tree decomposition* of a graph $G = (N, E)$ is a pair $\langle T, \chi \rangle$, where $T = (V, F)$ is a tree, and χ is a labeling function assigning to each vertex $p \in V$ a set of vertices $\chi(p) \subseteq N$, such that the following three conditions are satisfied:

1. for each node $b \in N$, there exists $p \in V$, such that $b \in \chi(p)$;
2. for each edge $(b, d) \in E$, there exists $p \in V$, such that $\{b, d\} \subseteq \chi(p)$; and
3. for each node b of G, the set $\{p \in V | b \in \chi(p)\}$ induces a connected subtree of T.

The width of $\langle T, \chi \rangle$ is the number $max_{p \in V}(|\chi(p)| - 1)$. The *treewidth* of G is the minimum width over all its tree decompositions.

So to phrase this in simpler terms, a tree decomposition of a graph is a tree where all nodes of the graph are mapped (non-injectively) to the tree nodes, and the co-occurring vertices of all graph edges are included in some tree node. Finally, if a graph vertex appears in any two tree nodes, it must be in all nodes in paths between them. This last property is also known as *running intersection property* or coherence. A tree decomposition is not unique. The width is characterized by the tree node with the most graph vertex, and the minimal width of any decomposition is the treewidth of the graph.

Intuitively, treewidth measures *how close the graph is to a tree.* For instance, a tree[4] has treewidth 1, and a k-clique[5] (which is very cyclic) has treewidth $k - 1$. Queries with $\text{tw} = 1$ are also called *acyclic.*

The importance of the shape of conjunctive queries becomes clear in the following result, linking the *treewidth (tw)* of the query's graph to the complexity of query evaluation.

[4]The decomposition is similar to the input tree, there are two nodes in each bag, one for the parent and one for the child.

[5]The nodes of the clique are a single bag in the decomposition, otherwise the conditions (2) and (3) of Definition 4.2.2 do not hold.

Theorem 4.2.3 (cfr. [CR97]; [GLS01]; [KV98])**.** *Let G be a graph and Q an eligible conjunctive query for which the canonical graph has treewidth k. Then it can be tested in time $|G|^{O(k)}|Q|^{O(k)}$ if Q returns a non-empty result on G.*

Since some queries do not have binary edges, the adequate model not a graph, but a hypergraph. The notion of treewidth can be extended from graphs to hypergraphs, which is called hypertreewidth.

Definition 4.2.4. A *hypertree decomposition* of a hypergraph $\mathcal{H}$ is a hypertree $\mathcal{HD} = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$, with λ as a function labeling the vertices of T by sets of hyperedges of $\mathcal{H}$, such that

1. $\langle T, \chi \rangle$ is a tree decomposition of $\mathcal{H}^a$;
2. for each $p \in \mathit{vertices}(T)$, $\chi(p) \subseteq \bigcup_{h \in \lambda(v)} h$. That is, all nodes in the χ labeling are covered by hyperedges in the λ labeling;
3. for each $p \in N$, $\mathit{vertices}(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$. Here, T_p denotes the subtree of T rooted at p. and $\chi(T_p)$ the set of all variables occurring in the χ labeling of this subtree. This means, that if p occurs in λ of a node, but not χ, then it must not occur in χ of the child nodes. This condition is also called *special condition* (or descendant condition).

The *width of a hypertree* (or hypertreewidth) is the cardinality of its largest λ label, i.e., $max_{p \in N}|\lambda(p)|$. By dropping the special condition, a hypertree is a *generalized hypertree decomposition* if all other conditions for a hypertree decomposition hold. Correspondingly, the width of a generalized hypertree decomposition is the *generalized hypertreewidth.*

[a] In Definition 4.2.2 (2), the set may now contain more than two elements.

The following example from [GMS09] shows a generalized hypertree decomposition and a hypertree decomposition of a hypergraph. Figure 4.7 shows the hypergraph, while decompositions are shown in Figure 4.8 and Figure 4.9.

Example 4.2.5. Hypergraph with hypertreewidth 3.

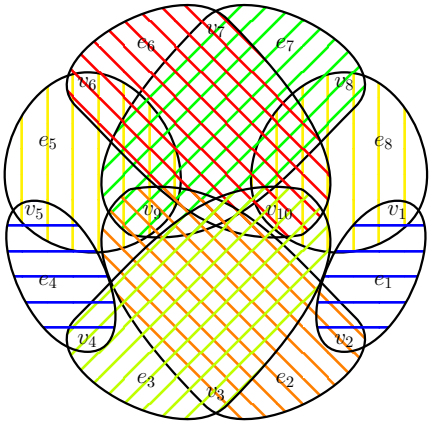


Figure 4.7: Complex hypergraph H_0 of a query.

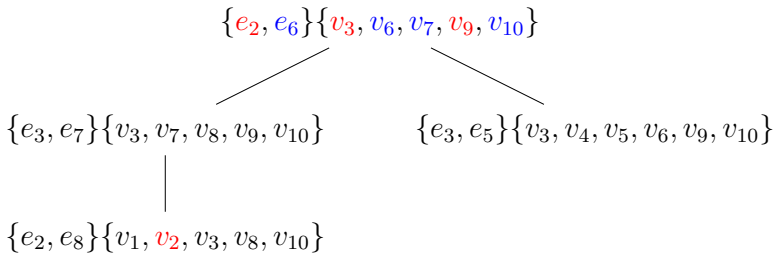


Figure 4.8: A *generalized* hypertree decomposition of H_0 with width 2.

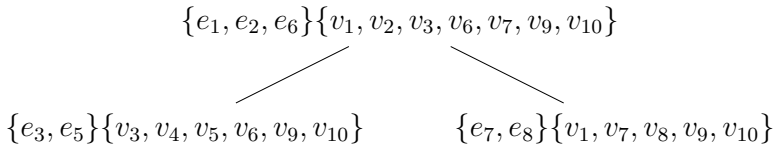


Figure 4.9: A hypertree decomposition of H_0 with width 3.

Notice that the generalized hypertree decomposition in Figure 4.8 violates the *special condition*: In the root node, e_2 in λ includes v_2, which does not appear in χ. But v_2 appears in χ of the leftmost leaf, which is not allowed in this case.

Because of the importance explained before in this section, the tree- and hypertreewidth of CQ, $\mathsf{CQ_F}$ $\mathsf{CQ_{OF}}$, and $\mathsf{CQ_{OFV}}$ queries were investigated and the question of acyclicity served as a basis for the investigation of graph shapes. In the terminology of Gottlob et al., the *treewidth* of the graphs of the queries and the *hypertreewidth* of the canonical hypergraphs of queries were investigated.

We will first look at treewidth and hypertreewidth for the Multi-Source Collection, and then we will turn to the Wikidata Collection.

Treewidth in Multi-Source Collection All shapes discussed in Section 4.2.1 have treewidth at most two. Forests (and all subclasses thereof) have treewidth one, whereas flowers and bouquets have treewidth two. The remaining queries were inspected by using the tool JDrasil[6] [BBE17] and three queries were discovered that had treewidth three (one such query is in Figure 4.10) and all others had treewidth two, see Table 4.3. From the treewidth perspective, it is interesting to note that many queries of treewidth two are also *flowers* or *bouquets* (Definition 4.2.1), which are a very restricted fragment.

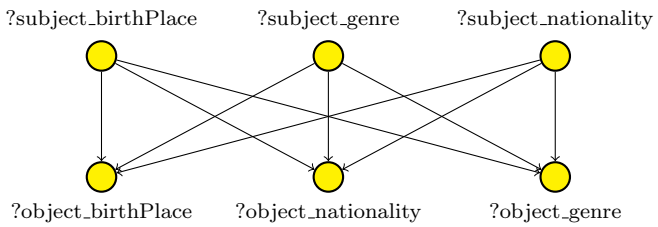


Figure 4.10: The DBpedia query exhibiting treewidth equal to 3.

Hypertreewidth in Multi-Source Collection Recall that only graphs of queries are considered, for which variables in the predicate position are not re-used elsewhere (if they occur at all). In $\mathsf{CQ_{OFV}}$, 58,782,592 (17,333,741) queries used a variable in a predicate position, or a filter, or Values operator of arity greater than two. Only for those queries their hypergraph structure is considered, without constants, to assess the cyclicity of these queries. Their hypertreewidth was determined with the tool `detkdecomp` [Sam18], which is based on [GS09]. Furthermore, the cyclicity of the hypergraphs is measured *without constants*, as it is usually done in the literature.

Our results are summarized in Table 4.8, which contains the hypertreewidth of queries from CQ, $\mathsf{CQ_F}$, $\mathsf{CQ_{OF}}$, and $\mathsf{CQ_{OFV}}$ that were not yet

[6] Available on `https://maxbannach.github.io/Jdrasil/`

analyzed in Section 4.2.1. Concerning CQ, all the remaining queries had hypertreewidth one, except for 68 (56) queries with hypertreewidth two, and eight queries with hypertreewidth three. In the largest fragment, $\mathsf{CQ}_{\mathsf{OFV}}$, there are 542,409 (242,941) such queries with hypertreewidth two, and nine with hypertreewidth three. So, especially in the fragment $\mathsf{CQ}_{\mathsf{OFV}}$, there is a significant portion of the queries that exhibits cyclicity, i.e., 8.03% of the unique queries.

The number of nodes in the hypertree decompositions that the tool returns is also of interest, since this number can be a guide for how well *caching* can be exploited for query evaluation [KEK17] (the higher the number, the better caching can be exploited). For the queries with hypertreewidth one, the number of nodes in the decompositions corresponds to their number of edges, which can already be seen in Figure 4.5. (Nevertheless, there are several hundred queries in $\mathsf{CQ}_{\mathsf{OFV}}$ queries with 100 or more nodes in their hypertree decompositions, the vast majority occurring in the DBpedia logs.) Finally, out of the queries with hypertreewidth two, 598 (465) had decompositions of size more than 10, going up to a maximum of 16. The $\mathsf{CQ}_{\mathsf{OFV}}$ queries of hypertreewidth three all had decompositions of size smaller than 10, except for one query in `DBpedia17` which had a decomposition of size 33.

Wikidata Collection *Hypergraphs* generalize graphs in the sense that they allow more than two nodes per edge. As such, the queries that were not suitable for graph shape analysis in Section 4.2.1, because they either went beyond *graph patterns*, or used Filter, Service, Bind, or Values constraints with arity three or more can be considered here. This amounts to a total of 1,915,550 (1,229,035) $\mathsf{CQ}_{\mathsf{OF}+}$ queries that were not yet analyzed in Section 4.2.1.

We keep the restriction on well-designed Optional constructs with interface width 1, since for these queries, there still is a correlation between the cyclicity of the hypergraph and complexity of query evaluation [Bar+18]. We do not consider queries with property paths in the hypergraph analysis.

A *hypergraph* is a pair $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is its finite set of nodes and $\mathcal{E} \subseteq 2^{\mathcal{V}}$ is a set of *hyperedges*. The *canonical hypergraph of a SPARQL pattern* P is defined as $\mathcal{E} = \{X \mid \text{there is a triple pattern } t \text{ in } P, \text{ such that } X \text{ is the set of blank nodes and variables appearing in } t\}$ and $\mathcal{V}$ is the union of the nodes in the edges in $\mathcal{E}$.

Using the tool `detkdecomp` [Sam18], we analyzed the hypergraphs of all $\mathsf{CQ}_{\mathsf{OF}+}$ queries for which the Optional constructs are well-designed and have interface width one. Overall, we found 590,005 (273,947) remaining queries with hypertreewidth two. All others new queries had hypertreewidth one.

	CQ				$\mathsf{CQ_F}$			
	#Valid	%Valid	#Unique	%Unique	#Valid	%Valid	#Unique	%Unique
$htw = 1$	4,137,042	100.00%	2,338,797	100.00%	5,162,377	95.42%	2,557,651	99.17%
$htw = 2$	68	0.00%	56	0.00%	248,050	4.58%	21,410	0.83%
$htw = 3$	8	0.00%	8	0.00%	8	0.00%	8	0.00%
Total new	4,137,118	100.00%	2,338,861	100.00%	5,410,435	100.00%	2,579,069	100.00%

	$\mathsf{CQ_{OF}}$				$\mathsf{CQ_{OFV}}$			
	#Valid	%Valid	#Unique	%Unique	#Valid	%Valid	#Unique	%Unique
$htw = 1$	26,680,385	99.07%	2,743,833	99.22%	26,725,649	98.01%	2,780,838	91.97%
$htw = 2$	249,126	0.93%	21,678	0.78%	542,409	1.99%	242,941	8.03%
$htw = 3$	8	0.00%	8	0.00%	9	0.00%	9	0.00%
Total new	26,929,519	100.00%	2,765,519	100.00%	27,268,067	100.00%	3,023,788	100.00%

Table 4.8: Hypertreewidth (*htw*) of the queries in Multi-Source Collection that were not discussed in Section 4.2.1, i.e. queries that use filter- or values conditions of arity three or more; or that re-use some variable in the predicate position elsewhere.

4.2.3 Quantitative Measuring of Shapes

In this section, a deeper characterization of the query shapes found in the logs is performed, by presenting various measures of these shapes. The focus is here is on chain, tree, and star-shaped queries, which are the most recurring shapes in our logs. Some measures are shared for certain shapes, while some are separate for a class. This is because some measures are not applicable for some shapes, while other measures do not make sense to measure, for instance, chains do not diverge and as such, measuring degrees is not needed, because all inner nodes have degrees two, while the ends have degree one. While most shapes are non-cyclic, later on, there will also be more insights about the cyclic queries the logs.

In the presentation, the results will be tagged to aid in quickly identifying the results: The Multi-Source Collection will be abbreviated as MS, while the Wikidata Collection will be annotated as WD.

To give an overview of what measures in shapes will be analyzed: Longest paths in chains, stars, and trees; maximum and average degrees in trees; minimum and maximum lengths of cycles.

Multi-Source Collection An immediate measure of the span of a query shape is the size of the longest (undirected) path in the query. Such a measure is readily applicable to chains, stars and tree-shaped queries. The size of the longest path for a tree-shaped query is the length of the longest path from one leaf to another leaf.

Example 4.2.6. For instance, consider the tree-shaped query in Figure 4.11. Observe that its longest path has length 7 (highlighted in bold). The same applies to star-shaped queries where the longest path is the path from one vertex to another traversing the central node of the star, whereas the longest path in a chain is the length of the chain itself.

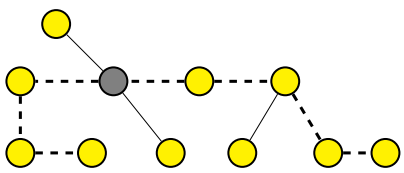


Figure 4.11: A tree-shaped query with longest path of length 7 (in bold and dashed) and maximal degree of nodes equal to 4 (for the grey node).

length	#V chain	%V chain	#V star	%V star	#V tree	%V tree
1	142,644,649	87.34%	-	-	-	-
2	16,185,787	9.91%	7,884,906	92.42%	-	-
3	3,880,284	2.38%	376,217	4.41%	59,537	8.00%
4	601,580	0.37%	264,287	3.10%	284,953	38.29%
5	1,970	0.00%	6,408	0.08%	14,167	1.90%
6	2,132	0.00%	136	0.00%	385,110	51.75%
7	1,011	0.00%	10	0.00%	436	0.06%
8	1,015	0.00%	8	0.00%	2	0.00%
9	4	0.00%	7	0.00%	0	0.00%
10–23	8	0.00%	11	0.00%	2	0.00%
total	163,318,440	100.00%	8,531,990	100.00%	744,207	100.00%

length	#U chain	%U chain	#U star	%U star	#U tree	%U tree
1	49,039,098	84.01%	-	-	-	-
2	6,853,199	11.74%	3,833,545	91.21%	-	-
3	2,400,853	4.11%	212,739	5.06%	17,213	15.56%
4	76,828	0.13%	155,883	3.71%	31,779	28.73%
5	1,333	0.00%	901	0.02%	12,752	11.53%
6	1,468	0.00%	50	0.00%	48,792	44.11%
7	1,009	0.00%	8	0.00%	79	0.07%
8	1,011	0.00%	8	0.00%	2	0.00%
9	3	0.00%	6	0.00%	0	0.00%
10–23	7	0.00%	7	0.00%	2	0.00%
total	58,374,809	100.00%	4,203,147	100.00%	110,619	100.00%

Table 4.9: Analysis of longest paths in chain, star, and tree queries (*Valid* and *Unique* queries) in Multi-Source Collection.

Longest Paths (MS) Table 4.9 reports the lengths of the longest paths in chain, tree, and star-shaped queries in the Multi-Source Collection. One can notice that the longest paths in chain and star queries are mostly small (significant percentages go up to size of the longest path equal to 3 for chain queries, and to 4 for star queries, respectively), whereas trees are somehow different. Their non-zero percentages characterize lengths of longest paths up to 6 for tree-shaped queries. In all shapes, there are some examples of queries with quite long paths (from length 10 to 23) and these are comparably higher in chains and stars than in tree-shaped queries.

degree	#V star	%V star	#U star	%U star
3	5,791,971	67.89%	3,173,041	75.49%
4	1,183,578	13.87%	406,272	9.67%
5	350,676	4.11%	191,479	4.56%
6	710,511	8.33%	228,573	5.44%
7	223,651	2.62%	68,179	1.62%
8	78,890	0.92%	55,056	1.31%
9	38,711	0.45%	25,152	0.60%
10–19	147,266	1.73%	53,067	1.26%
20–29	2,758	0.03%	2,077	0.05%
30–39	230	0.00%	192	0.00%
40–49	64	0.00%	51	0.00%
50–59	6	0.00%	6	0.00%
60–63	3,678	0.04%	2	0.00%
total	8,531,990	100.00%	4,203,147	100.00%

degree	#V tree	%V tree	#U tree	%U tree
3	401,873	54.00%	73,125	66.11%
4	26,154	3.51%	2,640	2.39%
5	279,092	37.50%	30,844	27.88%
6	31,258	4.20%	3,305	2.99%
7	5,367	0.72%	589	0.53%
8	375	0.05%	51	0.05%
9	47	0.01%	36	0.03%
10–19	39	0.01%	27	0.02%
20–29	2	0.00%	2	0.00%
total	744,207	100.00%	110,619	100.00%

Table 4.10: Maximal degree of nodes in star and tree queries (*Valid* (V) and *Unique* (U)) in Multi-Source Collection.

Maximum Degrees (MS) The next analysis of the shapes is focusing on the nodes with the maximal degree of nodes in star- and tree-shaped queries. In the example of a tree-shaped query in Figure 4.11, it can be seen that the maximal degree of nodes is equal to 4. Obviously, this measure is not informative for chain queries, since they are completely characterized by their length (and their vertices have a degree of a most 2). Table 4.10 shows the results for stars and tree-shaped queries.

The higher percentages of star queries have maximal degree of their ver-

#HD	#V tree	%V tree	#U tree	%U tree
2	59,537	8.00%	17,213	15.56%
3	281,184	37.78%	31,197	28.20%
4	14,348	1.93%	12,877	11.64%
5	365,318	49.09%	47,920	43.32%
6	23,811	3.20%	1,405	1.27%
7	7	0.00%	5	0.00%
9	1	0.00%	1	0.00%
11	1	0.00%	1	0.00%
total	744,207	100.00%	110,619	100.00%

Table 4.11: Number of high-degree nodes (#HD) in tree-shaped queries (*Valid* (V) and *Unique* (U)) in Multi-Source Collection.

tices equal to 3, whereas for tree-shaped queries, the majority has maximal degree equal to 3 or 5. The highest values of maximal degrees can be observed in stars more than in tree-shaped queries.

Tree Queries (MS) The next focus is on tree-shaped queries. To this end, the number of queries were computed that contained nodes with highest degrees. This measure is only applicable to tree-shaped queries and neither to stars (that always have a single node with a highest degree) nor to chains. The results are shown in Table 4.11. There, one can notice 49.09% (43.32%) of the tree-shaped queries have 5 high-degree vertices. Otherwise, 3 is the second most popular number for high-degree vertices covering 37.78% (28.20%). This means that an odd number of high-degree vertices is more popular. A high-degree vertex is a vertex that branches more than two times. There was also one unique query with 9 high-degree vertices and another unique one with 11 high-degree vertices.

Some values of high-degree nodes are aggregated because of their scarcity, this provides an easier overview. There is combined view of for degrees in queries of different shapes in Table 4.10 (maximal) and Table 4.12 (average) to provide a quick grasp of more details regarding the values of degrees.

AvgDeg	#V tree	%V tree	#U tree	%U tree
2–2.9	400,426	53.81%	61,955	56.01%
3–3.9	308,649	41.47%	44,358	40.10%
4–4.9	34,514	4.64%	4,027	3.64%
5–5.9	346	0.05%	160	0.14%
6–6.9	103	0.01%	52	0.05%
7–7.9	157	0.02%	58	0.05%
8–8.9	12	0.00%	9	0.01%
total	744,207	100.00%	110,619	100.00%

Table 4.12: Average degree of inner nodes (AvgDeg) in tree-shaped queries (*Valid* (V) and *Unique* (U)) in Multi-Source Collection.

Average Degrees (MS) Further investigating the tree shapes, in Table 4.12 the average degrees of inner nodes in these shapes is computed (again not applicable to chains and stars). Observer that the majority of inner nodes degrees stay between 2 and 4 on average.

Cycle Lengths (MS) Finally, the class of cyclic queries was investigated. The minimal and maximal lengths of all cycles in a query was measured. The cycle computation again considered the queries as undirected graphs and computed the cycle basis for such graphs. A cycle basis is formed from any spanning tree or spanning forest of the given graph, by selecting the cycles obtained by combining a path in the tree with a single edge outside the tree. This technique is described in [CGH95]. In order to keep the computation of cycle basis polynomial, an empirical bound (equal to 8) was set for the number of cycles that form the cycle basis. Thus the minimal and maximal cycle length of the discovered cycle basis of each query were counted. Table 4.13 (maximal) and Table 4.14 (minimal) report the results of this analysis for $\mathsf{CQ}_{\mathsf{OFV}}$ queries.

Free-Connex Acyclicity (MS) Another measure that was computed is the property of free-connex acyclicity for CQ, CQ_{F}, $\mathsf{CQ}_{\mathsf{OF}}$ and $\mathsf{CQ}_{\mathsf{OFV}}$. A conjunctive query is free-connex acyclic if it is acyclic and the set of its free variables[7] is a connex subset of the join tree of the query [BDG07]. The join tree of a query corresponds to the tree-structure of the acyclic hypergraph underlying the query. Free-connex acyclicity is interesting because it

[7]The free or distinguished variables of a query considered as a first-order propositional formula are the set of variables used as *output* in the formula.

Max	#Valid	#Unique
3	1,455,724	328,118
4	51,308	23,946
5	25,062	5,865
6	3,243	79
7	7	7
8	1	1
10	1	1
total	1,535,346	358,017

Table 4.13: Maximal cycle length in cyclic queries in Multi-Source Collection.

Min	#Valid	#Unique
3	1,456,037	328,347
4	51,023	23,739
5	25,048	5,853
6	3,230	70
7	7	7
10	1	1
total	1,535,346	358,017

Table 4.14: Minimal cycle length in cyclic queries in Multi-Source Collection.

characterizes the conjunctive queries for which certain kinds of efficient algorithms exist for enumerating their output [BDG07]; [IUV17] (under standard complexity-theoretical assumptions).

Table 4.15 shows the results by comparing the number of all conjunctive queries (including those that are not suitable for graph analysis and thus are not considered in Table 4.3) and the number of free-connex acyclic queries found in the Multi-Source Collection. Notice that the latter are abundant in all the fragments CQ, $\mathsf{CQ_F}$, $\mathsf{CQ_{OF}}$ and $\mathsf{CQ_{OFV}}$. For a cross comparison, the hypertreewidth of all the conjunctive queries in the logs is shown (and not only those reported in Table 4.3). Observe that all the CQs in our logs have *htw* less or equal to 3.

	CQ				$\mathsf{CQ_F}$			
	#Valid	%Valid	#Unique	%Unique	#Valid	%Valid	#Unique	
FCA	117,669,790	96.14%	36,786,611	93.00%	152,870,355	93.98%	53,393,254	91.19%
$htw \leq 1$	118,245,559	96.61%	37,212,024	94.08%	157,167,354	96.63%	55,954,365	95.56%
$htw \leq 2$	122,391,781	100.00%	39,555,004	100,00%	162,654,843	100.00%	58,554,583	100.00%
$htw \leq 3$	122,391,794	100.00%	39,555,014	100.00%	162,654,856	100.00%	58,554,593	100.00%
Total	122,391,794	100.00%	39,555,014	100.00%	162,654,856	100.00%	58,554,593	100.00%

	$\mathsf{CQ_{OF}}$				$\mathsf{CQ_{OFV}}$			
	#Valid	%Valid	#Unique	%Unique	#Valid	%Valid	#Unique	
FCA	160,545,014	80.02%	55,059,069	84.89%	163,203,235	58.15%	57,331,127	73.42%
$htw \leq 1$	170,732,618	85,09%	55,954,365	86.27%	173,407,077	61.78%	63,109,343	80.82%
$htw \leq 2$	200,641,878	100.00%	64,857,879	100.00%	280,672,718	100.00%	78,088,783	100.00%
$htw \leq 3$	200,641,891	100.00%	64,857,889	100.00%	280,672,732	100.00%	78,088,794	100.00%
Total	200,641,891	100.00%	64,857,889	100.00%	280,672,732	100.00%	78,088,794	100.00%

Table 4.15: Free-connex acyclicity (FCA) and *htw* of all the CQs in Multi-Source Collection logs.

Shapes in Wikidata Collection We now turn to the quantitative analysis of shapes in the Wikidata Collection. For this, it is especially interesting to compare organic queries to robotic queries. Additionally, the results are also looking at the *Valid* and *Unique* variants of the data sets, because this can show that there are some very distinct queries in the data sets, and that some high concentrations in *Valid* logs are in fact duplicates.

depth	#O V	%O V	#R V	%R V
2	107,436	51.83%	125,277,683	79.21%
3	80,736	38.95%	29,562,694	18.69%
4	15,812	7.63%	3,195,157	2.02%
5	2,412	1.16%	113,203	0.07%
6	239	0.12%	1,994	0.00%
7	644	0.31%	0	0.00%
8	6	0.00%	0	0.00%
9	7	0.00%	0	0.00%
10	0	0.00%	164	0.00%
total	207,292	100.00%	158,150,895	100.00%

depth	#O U	%O U	#R U	%R U
2	50,908	48.41%	24,688,374	92.24%
3	45,495	43.27%	1,919,179	7.17%
4	7,356	7.00%	151,649	0.57%
5	1,161	1.10%	4,971	0.02%
6	124	0.12%	27	0.00%
7	97	0.09%	0	0.00%
8	4	0.00%	0	0.00%
9	7	0.01%	0	0.00%
10	0	0.00%	1	0.00%
total	105,152	100.00%	26,764,201	100.00%

Table 4.16: Analysis of longest paths in chain queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

Longest Paths in Chains (WD) Table 4.16 depicts the longest paths in chain queries. Chain queries of length 2 make up the overwhelming majority of robotic chains, with almost 80% for *Valid* logs and going up even more to 90% for *Unique* logs. The rest of robotic queries has length 3 and only a negligible amount (around 1-2%) of queries being longer. Surprisingly, the

depth	#O V	%O V	#R V	%R V
2	53,581	64.27%	8,509,904	62.50%
3	21,050	25.25%	4,546,615	33.39%
4	7,813	9.37%	551,265	4.05%
5	850	1.02%	6,039	0.04%
6	62	0.07%	1,490	0.01%
7	10	0.01%	1,202	0.01%
8	2	0.00%	0	0.00%
9	5	0.01%	0	0.00%
total	83,373	100.00%	13,616,515	100.00%

depth	#O U	%O U	#R U	%R U
2	27,976	66.54%	2,056,510	64.93%
3	10,270	24.43%	1,000,269	31.58%
4	3,489	8.30%	106,350	3.36%
5	257	0.61%	3,380	0.11%
6	38	0.09%	356	0.01%
7	10	0.02%	214	0.01%
8	2	0.00%	0	0.00%
9	5	0.01%	0	0.00%
total	42,047	100.00%	3,167,079	100.00%

Table 4.17: Analysis of longest paths in star queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

longest chain is a robotic query of length 10 in the *Unique* logs which appears 164 times in the *Valid* logs. Organic logs are a bit more distributed, with only around 50% using length 2, around 40% using length 3, and around 7% using length 4. The rest quickly declines to 1% and less.

Longest Paths in Stars (WD) The longest paths in in star queries are shown in Table 4.17. Star-shaped queries look very similar across the board when looking at organic and robotic queries and *Valid* and *Unique* logs. Roughly two thirds use only depth 2, which is the minimal star depth. The second largest depth 3 is used around 25% for organic stars, and about one third for robotic queries. Then 8-9% (for *Unique*, resp. *Valid*) organic queries use depth 4, while the robotic queries and the rest of the organic queries use negligible amounts of queries with higher depths. The maximum depth goes up to 8 and 9 for a few organic queries, which surprisingly seem to be unique.

depth	#O V	%O V	#R V	%R V
3	21,885	56.06%	69,110	58.65%
4	9,369	24.00%	40,161	34.08%
5	7,375	18.89%	8,306	7.05%
6	293	0.75%	254	0.22%
7	105	0.27%	4	0.00%
8	4	0.01%	0	0.00%
9	0	0.00%	2	0.00%
10	1	0.00%	0	0.00%
11	3	0.01%	0	0.00%
12	1	0.00%	0	0.00%
total	39,036	100.00%	117,837	100.00%

depth	#O U	%O U	#R U	%R U
3	5,723	40.87%	6,753	44.23%
4	3,445	24.60%	3,502	22.94%
5	4,574	32.67%	4,966	32.53%
6	205	1.46%	42	0.28%
7	48	0.34%	3	0.02%
8	2	0.01%	0	0.00%
9	0	0.00%	1	0.01%
10	1	0.01%	0	0.00%
11	3	0.02%	0	0.00%
12	1	0.01%	0	0.00%
total	14,002	100.00%	15,267	100.00%

Table 4.18: Analysis of longest paths in tree queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

Longest Paths in Trees (WD) After going from chains to stars, finally Table 4.18 depicts the longest paths in tree queries. Very few tree-shaped queries exceed a depth of 7. There seems to be one robotic query with depth 9 which occurs two times, and some organic queries with length 8-12 which in some cases appear to be unique or with a very low number of duplicates. If one looks at the deduplicated *Unique* logs, it is interesting to note that the organic and robotic queries are quite close: Around 41-44% use depth 3, 23-25% use depth 4, and 33% use depth 5. This changes a bit if one focuses on the *Valid* logs with duplicates, there are both organic and robotic queries use 56-58% of depth 3, but organic queries have 24% of depth 4 and 19% of

max degree	#O V	%O V	#R V	%R V
3	6,891	17.65%	60,960	51.73%
4	6,544	16.76%	15,377	13.05%
5	18,255	46.76%	5,523	4.69%
6	2,060	5.28%	19,889	16.88%
7	550	1.41%	1,814	1.54%
8	635	1.63%	279	0.24%
9	1,719	4.40%	11,570	9.82%
10	2,000	5.12%	8	0.01%
11	265	0.68%	528	0.45%
12	44	0.11%	798	0.68%
13	2	0.01%	0	0.00%
14	2	0.01%	0	0.00%
15	0	0.00%	214	0.18%
16	1	0.00%	0	0.00%
17	62	0.16%	877	0.74%
18	2	0.01%	0	0.00%
19	4	0.01%	0	0.00%
total	39,036	100.00%	117,837	100.00%

Table 4.19: Maximum degree in tree queries (Robotic (R) and Organic (O), *Valid* (V)).

depth 5, while robotic logs use 34% of depth 4 and 7% of depth 5. Overall, robotic queries are a bit simpler, and there are a few very unique deeper queries in the organic queries. This is what one would expect.

Maximum Degrees in Trees (WD) The maximum degree of nodes in tree queries are depicted in Table 4.19 (*Valid*) and Table 4.20 (*Unique*). On first glance, most tree-shaped queries have a maximum degree of less than 5. In more detail, it is noticeable that organic queries seem to have a majority of queries with degree 5, but this difference shrinks if one goes from *Valid* to *Unique* queries.

For robotic queries, more than 50% are the simplest trees with degree 3, but this shrinks dramatically and shifts to the degree 4 if one goes from *Valid* to *Unique*.

The highest degree goes up to 19 for some organic queries, which is a bit higher than 17, which is the highest degree for robotic queries. It is interesting that there seem to be some degrees that are used more often after skipping some degrees, this is true even after deduplicating the queries. For

max degree	#O U	%O U	#R U	%R U
3	2,812	20.08%	4,953	32.44%
4	3,645	26.03%	6,242	40.89%
5	5,196	37.11%	2,792	18.29%
6	583	4.16%	502	3.29%
7	246	1.76%	49	0.32%
8	264	1.89%	24	0.16%
9	609	4.35%	438	2.87%
10	324	2.31%	2	0.01%
11	242	1.73%	22	0.14%
12	33	0.24%	33	0.22%
13	2	0.01%	0	0.00%
14	2	0.01%	0	0.00%
15	0	0.00%	63	0.41%
16	1	0.01%	0	0.00%
17	38	0.27%	147	0.96%
18	2	0.01%	0	0.00%
19	3	0.02%	0	0.00%
total	14,002	100.00%	15,267	100.00%

Table 4.20: Maximum degree in tree queries (Robotic (R) and Organic (O), *Unique* (U)).

example, while degree 17 occurs in some notable numbers, this is not the case for queries after degree 12.

Maximum Degrees in Stars (WD) After examining tree queries, Table 4.21 shows the maximum degree of nodes in star queries. Star-shaped queries are quite similar for organic and robotic queries. Over 50% of organic queries and over 60% of robotic queries have degree 3, with the second largest degree being 4 then declining rapidly, with the decline for robotic queries being larger. There is no pronounced difference when going from *Valid* to *Unique* logs.

Interestingly, star-shaped queries seem to have higher degrees than tree-shaped queries. They go up to degree 27. While they this high, there are no notable observations when looking at degrees after 5.

max degree	#O V	%O V	#R V	%R V
3	46,048	55.23%	8,461,436	62.14%
4	18,017	21.61%	2,527,507	18.56%
5	7,426	8.91%	281,542	2.07%
6	4,504	5.40%	75,436	0.55%
7	2,312	2.77%	77,571	0.57%
8	2,270	2.72%	63,218	0.46%
9	270	0.32%	3,356	0.02%
10	350	0.42%	18,466	0.14%
11	709	0.85%	66,290	0.49%
12	990	1.19%	125,246	0.92%
13-20	473	0.57%	655,855	4.82%
21-25	3	0.00%	1,260,514	9,26%
27	1	0.00%	78	0.00%
total	83,373	100.00%	13,616,515	100.00%

max degree	#O U	%O U	#R U	%R U
3	24,391	58.01%	2,108,835	66.59%
4	9,082	21.60%	451,628	14.26%
5	4,051	9.63%	33,750	1.07%
6	1,963	4.67%	7,650	0.24%
7	812	1.93%	28,850	0.91%
8	740	1.76%	29,970	0.95%
9	212	0.50%	1,464	0.05%
10	235	0.56%	5,828	0.18%
11	113	0.27%	9,905	0.31%
12	110	0.26%	71,974	2.27%
13-20	335	0.80%	202,481	6.39%
21-25	2	0,00%	214,741	6.78%
27	1	0.00%	3	0.00%
total	42,047	100.00%	3,167,079	100.00%

Table 4.21: Maximum degree in star queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

#HD	#O V	%O V	#R V	%R V
2	35,252	90.31%	97,279	82.55%
3	2,148	5.50%	20,428	17.34%
4	249	0.64%	128	0.11%
5	9	0.02%	0	0.00%
6	353	0.90%	2	0.00%
7	558	1.43%	0	0.00%
8	467	1.20%	0	0.00%
total	39,036	100.00%	117,837	100.00%

#HD	#O U	%O U	#R U	%R U
2	12,027	85.89%	14,685	96.19%
3	781	5.58%	570	3.73%
4	108	0.77%	10	0.07%
5	7	0.05%	0	0.00%
6	328	2.34%	2	0.01%
7	518	3.70%	0	0.00%
8	233	1.66%	0	0.00%
total	14,002	100.00%	15,267	100.00%

Table 4.22: Number of high-degree nodes in tree-shaped queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

Inner Nodes (WD) Table 4.22 depicts the number of inner nodes in tree queries. Very few tree-shaped queries contain nodes that have a highest degree node larger than 3. Organic queries still manage to go up to 5-8 for a fair number of queries, but there are only 2 robotic queries that go up to 6. Still, 85-90% (*Unique*, resp. *Valid*) of organic queries only go up to 2.

For *Valid* robotic queries, with 83% going up to 2, and 17% going up to 3, this suggests that robotic queries are more diverse. But if one goes from *Valid* to *Unique* logs, this changes drastically, with over 96% of robotic queries going up to only 2. So there is a very low number of uniquely more complex robotic queries.

range	#O V	%O V	#R V	%R V
2-2.9	2,422	6.20%	1,094	0.93%
3-3.9	17,426	44.64%	99,275	84.25%
4-4.9	17,336	44.41%	1,677	1.42%
5-5.9	1,593	4.08%	13,185	11.19%
6-6.9	128	0.33%	189	0.16%
7-7.9	42	0.11%	776	0.66%
8-8.9	5	0.01%	528	0.45%
9-9.9	16	0.04%	236	0.20%
10-10.9	13	0.03%	650	0.55%
11-11.9	55	0.14%	227	0.19%
total	39,036	100.00%	117,837	100.00%

range	#O U	%O U	#R U	%R U
2-2.9	960	6.86%	106	0.69%
3-3.9	9,412	67.22%	13,630	89.28%
4-4.9	3,094	22.10%	766	5.02%
5-5.9	352	2.53%	480	3.14%
6-6.9	90	0.64%	20	0.13%
7-7.9	32	0.23%	11	0.07%
8-8.9	5	0.04%	22	0.14%
9-9.9	14	0.10%	85	0.56%
10-10.9	10	0.07%	100	0.66%
11-11.9	33	0.24%	47	0.31%
total	14,002	100.00%	15,267	100.00%

Table 4.23: Average degree of inner nodes in tree-shaped queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

Average Degrees (WD) The average degree of tree queries is shown in Table 4.23. The average degree of inner nodes of tree-shaped queries for organic logs has the highest concentration in the ranges from 3-3.9 and 4-4.9 with around 44% in unique logs. If one goes from *Valid* to *Unique* logs, this shifts to 67% to the lower range and 22% to the higher range. There is a small amount of organic queries above and below these ranges, but ranges higher than 6 are very rare. These ranges are also very rare for robotic queries. But for robotic queries, there are almost no queries in the 4-4.9 range, instead 84% are in the lower 3-3.9 range, and 11% are in the higher 5-5.9 range. This changes when going the deduplicated logs, then 89% are in the range above 3,

length	#O V	%O V	#R V	%R V
3	742	52.92%	3,443	27.48%
4	602	42.94%	8,979	71.67%
5	45	3.21%	82	0.65%
6	9	0.64%	24	0.19%
7	3	0.21%	0	0.00%
10	1	0.07%	0	0.00%
total	1,402	100.00%	12,528	100.00%

length	#O U	%O U	#R U	%R U
3	382	58.68%	92	2.12%
4	223	34.25%	4,249	97.81%
5	33	5.07%	1	0.02%
6	9	1.38%	2	0.05%
7	3	0.46%	0	0.00%
10	1	0.15%	0	0.00%
total	651	100.00%	4,344	100.00%

Table 4.24: Maximal cycle length in cyclic queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

5% are in the range above 4, and 3% are in the range above 5. This suggests that there seem to be a larger number of duplicates in the 4-4.9 range, but still, most queries (both organic and robotic), especially after deduplication, are in the 3-3.9 range.

Cycle Lengths (WD) Table 4.24 depicts the length of the longest cycle in queries. The maximum length of cycles of *Unique* robotic queries almost solely consists of cycles of length 4. It is interesting that the robotic queries seem to have a higher concentration of marginally larger length of 4 instead of the minimum length of 3. In *Valid* logs, organic queries still have more than 50% of only length 3, and 43% of length 4.

There is a minor shift going from *Valid* to *Unique* logs for organic logs, but for robotic logs, it goes from 27% for length 3, and 72% for length 4 to 98% for length 4.

Finally, longer cycles are very rare for robotic queries. There is one unique query with length 5, and 2 unique queries with length 6. On the other hand, there are over 50 queries in the organic logs over length 4, going up to a

length	#O V	%O V	#R V	%R V
3	751	53.57%	3,443	27.48%
4	607	43.30%	9,003	71.86%
5	35	2.50%	82	0.65%
6	5	0.36%	0	0.00%
7	3	0.21%	0	0.00%
10	1	0.07%	0	0.00%
total	1,402	100.00%	12,528	100.00%

length	#O U	%O U	#R U	%R U
3	390	59.91%	92	2.12%
4	227	34.87%	4,251	97.86%
5	25	3.84%	1	0.02%
6	5	0.77%	0	0.00%
7	3	0.46%	0	0.00%
10	1	0.15%	0	0.00%
total	651	100.00%	4,344	100.00%

Table 4.25: Minimal cycle length in cyclic queries (Robotic (R) and Organic (O), *Valid* (V) and *Unique* (U)).

length of 10 even. This is significant, because there are almost 10 times more robotic queries in this comparison. So again, as expected, it seems that there are more unique complex queries in the organic subset.

In addition to the maximum length, the minimum length of cycles in queries is shown in Table 4.25. The minimum cycle length of queries seems to be very close the maximum cycle length. This could suggest that there may not be many cycles in queries, or that they have the same length and the shape of the query could be very regular. Indeed, in the exploration of queries, there seem to have been many shapes that are very regular. For example, there are many queries that have a central node and regular looking cyclic parts attached to this node.

Free-Connex Acyclicity (WD) Table 4.26 shows the free-connex acyclicity in the CQ fragment of Wikidata Collection. As can be seen, the coverage in robotic queries is over 99%, while it is only a bit over 91% for organic queries. These results are to be similar to the results from the Multi-Source Collection, considering that organic and robotic queries are not separated there.

Log	#Organic	%Organic	#Robotic	%Robotic
Valid	80,420	92.97%	82,249,379	98.52%
Unique	33,414	91.04%	22,886,318	99.37%

Table 4.26: Free-Connex Acyclicity in the Wikidata Collection, showing results for *Valid* and *Unique*.

5

Inter-Query Analysis

In the following sections, we turn to quite different topics compared to earlier sections. Before, the focus was on isolated queries, and neither the order of queries log files was of no importance, nor where some structural properties that are lost in the analysis like the order of conjunctions or other idiosyncrasies that are lost during normalization. There are connections between queries in logs, which is the focus in the following.

5.1 Evolution of Queries over Time: Streaks

In a typical usage scenario of a SPARQL endpoint, a user queries the data and gradually refines the query until the desired result is obtained. In this section, an analysis is performed to determine to which extent such behavior occurs. The results show that, in certain contexts, it can be interesting to investigate optimization techniques for sequences of similar queries.

For the following purposes, a *query log* can be considered to be an ordered list of queries $q_1, \ldots, q_n$. We introduce a new notion called a *streak*, which intuitively captures a sequence of similar queries within close distance of each other. To this end we assume the existence of a *similarity test* between two queries.

Definition 5.1.1. Two queries q_i and q_j with $i < j$ *match* if (1) q_i and q_j are similar and (2) no query $q_{i'}$ with $i < i' < j$ is similar to q_i. A *streak* (with window size w) is a sequence of queries $q_{i_1}, \ldots, q_{i_k}$, such that, for each $\ell = 1, \ldots, k-1$, we have that $i_{\ell+1} - i_\ell \leq w$ and $q_{i_{\ell+1}}$ matches q_{i_ℓ}.

Notice that it is theoretically possible for a query to belong to multiple streaks. E.g., it is possible that q_1 and q_2 do not match, but query q_3 is

sufficiently similar to both. In this case, q_3 belongs to both streaks q_1, q_3 and q_2, q_3.

Generally calculating streaks can become very expensive, because each new query that gets examined must be tested against *all* ongoing streaks. The window size is important here, because it limits the number of ongoing streaks. There can be at most w ongoing queries, which would create streaks of length 1; otherwise there would exist a query in between matching queries, which would contradict Definition 5.1.1. Therefore, we need to track at most w streaks. Also note that we only need to store the last query in a streak for comparison with a new query. These two observations are very helpful for performing a streak analysis more efficiently.

For this study a curated subset of Multi-Source Collection is used. In Section 5.2 it will explained why Wikidata Collection is less suitable for this investigation due to its anonymization. In the present study, Levenshtein distance will be used as a similarity test. The Levenshtein distance [Lev66] is a String Edit Distance (SED) to measure the dissimilarity if two strings.

Definition 5.1.2. The edit distance of two string s_1 and s_2 is the minimum number of edit operations (insert, delete, substitute) to transform s_1 into s_2. The *normalized* string edit distance is obtained by dividing the string edit distance by $max(len(s_1), len(s_2))$.

This number is between 0 and 1 and been be interpreted as percentage, it measures dissimilarity (distance), but by inverting it, it measures similarity.

More precisely, two queries are *similar* if the Levenshtein distance of their strings, after removal of namespace prefixes, is at most 25%. Furthermore, namespace prefixes were removed (or rather resolved) prior to measuring their Levenshtein distance, because they can introduce superficial similarity. As such, queries are required to be at least 75% similar starting from the first occurrence of the keywords Select, Ask, Construct, or Describe. After some experimentation with some sample logs, a window size of 30 was chosen. This window size seemed to yield adequate performance and seemed to yield feasible results.

Streak length	*#DBP'14*	*#DBP'15*	*#DBP'16*
1–10	42,272	167,292	199,375
11–20	3,732	24,001	37,402
21–30	2,425	4,813	17,749
31–40	884	667	5,849
41–50	283	162	1,998
51–60	88	40	711
61–70	27	8	322
71–80	15	4	129
81–90	5	1	47
91–100	5	0	27
>100	4	0	24

Table 5.1: Length of streaks in three single-day logs.

Streak Length Since the discovery of streaks is extremely resource-consuming, only streaks in *three* randomly selected log files were analyzed. The selection was made from `DBpedia14`, `DBpedia15`, and `DBpedia16`, because those were some of the larger and more interesting logs from results from studies in the preceding sections. The sizes of these log files, each reflecting a single day of queries of the endpoint, were 273MiB, 803MiB, and 1004MiB respectively.

For the ordering of the queries, one could simply consider the ordering in the log files, since the logs are sorted over time.

The results on streak length are in Table 5.1. Using a window size of 30, the longest streak found had length 169 and was in the 2016 log file. When increasing the window size, it was still possible to obtain longer streaks. This likely means that a more refined analysis on the encountered streaks can be carried out when tuning the window size and deriving more complex metrics on the similarity of the queries within each streak.

Max #Triples	*#Streaks*	*Max #Triples*	*#Streaks*
1	130,706	13–20	9,509
2	41,811	21–30	544
3	34,081	31–40	233
4	9,990	41–50	86
5	3,325	51–60	44
6	1,733	61–70	32
7	8,465	71–80	17
8	10,604	81–90	11
9	7,837	91–100	3
10	1,080	101–110	9
11	51,521	> 110	7
12	43,819		

Table 5.2: Largest query occurring in streaks.

Evolution of Size and Structure In addition to the length of streaks, it was also investigated how the number of triples and structure of queries in streaks change over time. To this end, queries needed to be parsed in streaks, which was not the case before, it was sufficient to only regard the item of the streak as a string. The three log files contain a combined amount of 510,361 streaks. Out of these streaks, 321,042 have at least two queries and 234,627 additionally have at least one query that parses. Remarkably, in the latter set, only 1,402 streaks have an erroneous (i.e. not parsable) query. Here, 1,202 have an erroneous query followed by a correct one, and 789 have a correct query followed by an erroneous one.

Next the number of triples of queries in streaks were investigated. There are 355,466 streaks for which have at least one parsable query that contains at least one triple.[1] Table 5.2 contains, for each of the 355,466 streaks, what is the maximal number of triples in any of its queries. This number seems to be quite stable: There are only 3,915 streaks, in which this number changes during the streak.

[1]For 88,201 streaks, all queries had an empty body. Another 31 streaks had a non-empty body, containing no triples.

Shapes	*#Streaks*
containing *chn*	148,632
consisting only of *chn*	147,106
containing *bt*	39,839
consisting only of *bt*	39,810
containing *cyc*	526
consisting only of *cyc*	493
containing *bt* and *cyc*	12
consisting only of *bt* or *cyc*	40,315
containing *bt* and *chn*	2
consisting only of *bt* or *chn*	186,918
containing *chn* and *cyc*	21
consisting only of *chn* or *cyc*	147,620
consisting only of *chn*, *bt*, or *cyc*	187,444

Table 5.3: Structures of queries appearing in the same streak (*chn* = chain, *bt* = 'branching tree', i.e., tree that is not a chain, *cyc* = cyclic).

Table 5.3 contains results on the shapes of queries in streaks. The only shapes that were considered were: chain queries, trees that branch (and therefore are no chains), and cyclic queries, that is, queries that contain a cycle. Table 5.3 contains, for each subset S of these three shapes, the number of streaks that *contain only shapes from* S and the number of streaks that *consist only of shapes from* S.

Interestingly, there was a correlation between streak length and query shape and size. For instance, out of the 526 streaks that contain a cyclic query, 472 (89.73%) only consist of a single query. This strongly contrasts the entire log, where only 189,319 streaks (37.10%) consist of a single query. Similarly, there are 1,378 streaks that contain a query of at least 16 triples, but 1,332 of these streaks (96.66%) only have a single query. This suggests that highly complex queries are less likely to occur in longer streaks. Recall again that the data sets used for this study only consisted of DBpedia query logs for *three days*, which is a very small sample.

5.2 Query Similarity Search

During the development of the analysis, there were times such as in Section 4.2, it became clear that is would be useful to locate the exact provenance of some queries, and to locate similar queries. This tool proved to be very useful for the purpose of the studies that were performed in the previous sections. In general, massive query logs are valuable sources of information as long as they are usable for exploration and analysis. For purposes of the study of logs, as well as in order to improve the usability of the logs, a query similarity search facility was designed and implemented. This code is made publicly available and can be used by other people for their purposes (see Chapter 6). The principles and effectiveness of this facility is discussed in the following.

Edit Distance: From Strings to Trees As a first observation, a string similarity search with String Edit Distance (SED) as discussed in Section 5.1 runs into problems with the logs from the Wikidata Collection. They have been modified by an anonymization process prior to their release. This means that large elements of their strings were completely replaced, and various normalization techniques also changed the string representation of a query. Hence, it is highly likely that one cannot find a query by simply searching for its exact original string, or by trying to approximate it with a string similarity measure.

To overcome this problem a new different approach to the problem was taken: Instead of just using the query string, the processed query structure is used as model for comparing queries in terms of their similarity. This structure is a tree, and with this a Tree Edit Distance (TED) can be used.

This choice solves several problems when comparing queries: comments, prefixes, whitespace, normalization measures, and variable names do not affect the structure of a query and do not influence the computation of the query similarity measure. This is especially important with the Wikidata logs, because specifically in this case, in their anonymization, all variable names are renamed to uniform names with ascending numbers. Therefore, if a new variable in the query is placed before other variables, it will shift the names of *all* subsequent variables. The latter could potentially trigger a large modification if a variable with a shifted number is used in several places in the remainder of the query statement, although the change could have occurred much more locally in another place. The anonymization also removes all prefixes and inlines them, therefore the impact of changing a single IRI could become large depending on the length of the IRI in case of adoption of a SED measure.

Query AST To enable a structural search on Wikidata queries, the abstract syntax tree (AST) of a query is translated into a tree structure as follows. The *query type* and the *solution modifier* of the query respectively become a leaf node in the tree (with a value for the latter), whereas the *pattern* P is a subtree instantiated triple t with three leaves (from $\langle s, p, o \rangle$), or a property path pattern, or a subtree rooted in And, Filter, Union, Optional, Graph whose respective patterns are also subtrees, or, recursively, the subtree of a query Q.

Figure 5.1 exemplifies the translation of the following Wikidata query from the logs into the AST:

```
SELECT * WHERE {
  ?item wdt:P31 wd:Q146 .
  SERVICE wikibase:label {
    bd:serviceParam
    wikibase:language
    "[AUTO_LANGUAGE],en"
  }
}
```

Listing 5.1: SPARQL query

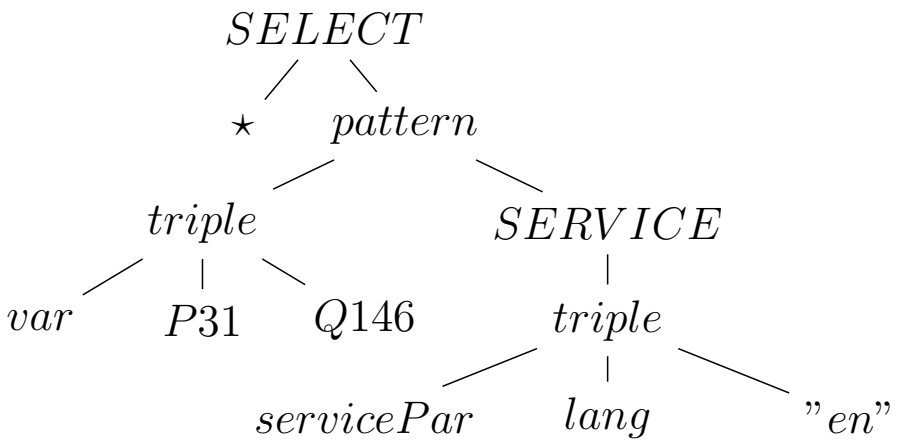


Figure 5.1: Abstract Syntax Tree of an example query.

Notice that in Figure 5.1, the actual instantiation of a triple variable name is ignored by labeling it *var*, as are the variable names that are used in the target list of the Select clause for the same reason. In the figure, the namespace prefixes are omitted for brevity, furthermore, they do not matter,

for identity of IRIs in the matching, they are completely resolved and have to match exactly. For the TED algorithm, each node can be given a weight contributing to the total sum of the AST that is used to calculate the TED. In particular, in the above AST for the example query, all nodes have equal weight (= 1, meaning that no specific bias is introduced). Adjustments of the weights are of course possible, targeting a specific user requirement.

Example 5.2.1. The example AST tree in Figure 5.1 has thus a total weight of 12, this being the sum of the weight of its nodes. Within the logs, a query almost identical to the above example was found that only changed the object *wd:Q146* to *wd:Q11538*. This renaming of a single node in a triple would result in a total similarity of 0.92.

Renaming two nodes in a triple would result in a similarity of 0.83. Adding a single triple anywhere, for example, outside the Service clause, would result in a total weight of 16. Such a change amounting to a total weight of 4 would result in a similarity of 0.75. Conversely, this can be seen as a removal by starting with a tree exhibiting the additional triple and deleting it.

As for the implementation of the TED algorithm, an available implementation of the APTED algorithm [PA15a]; [PA15b] was chosen, because the implementation was easy to embed into the software, it offered the option to use different weights for nodes, and it yielded good performance results when tested.

To calculate the similarity between an initial query Q and a new query Q_2, the size of the largest AST is taken and divided by the calculated TED. This value is inverted by subtracting it from 1.0 in order to measure similarity instead of dissimilarity. When collecting matches in a collection of queries (within the same log file), a maximum threshold of 0.75 was used. The reasoning is: A threshold of 0.5 would mean that two queries are equally similar and dissimilar, so the step to 0.75 is right in the middle between this and being classified as identical (which corresponds to similarity equal to 1.0). Based on manually inspecting samples, this already seems to be be a high threshold, and this method based on TED yielded more predictable results than a method based on SED.

To illustrate this: A typical Wikidata entity has a string length of about 36. Changing the entity number are 2-4 edits. Even if all triples are changed, this results in almost a 0.9 SED-similarity for the query, while even a two node change with TED in a small query would be classified as major change. Conversely, adding an additional Optional has an impact of only 4 with TED,

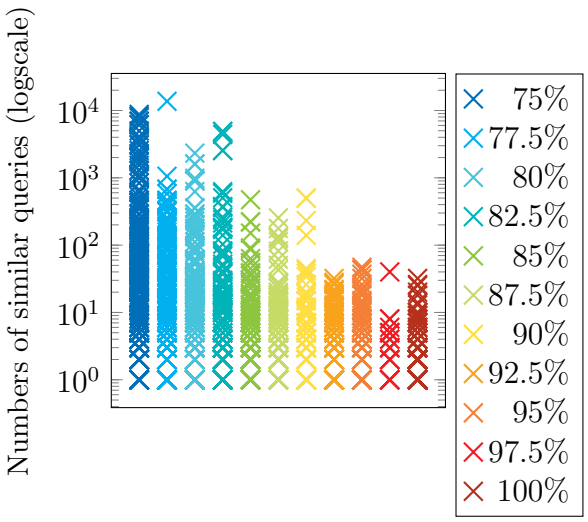


Figure 5.2: Number of queries that exhibit a similarity degree between 75% and 100% in Wikidata logs compared to examples.

while the SED changes by at least *158* if typical Wikidata entities are used, and this already assumes no extra white space usage.

Application on Wikidata Collection In order to measure the structural similarity of Wikidata queries as test, the focus was put on the OrganicOK query logs by considering their unique version only (since exact matches can just be detected by examining the strings)[2]. Since we needed seed queries for which we want to find similar queries, the well-known set of online Wikidata examples from [Wik18] come to mind. Via scripting, a set of 412 queries could be extracted and parsed by Jena.[3]

It turns out that matches for most online example queries are found in the Wikidata Collection. Only 28 queries could not be matched, while the rest had at least one match above the threshold.

Figure 5.2 shows (in log scale) the number of similar queries that were found in the OrganicOK unique query logs in the order of ascending similarity. Some outliers can be observed in several buckets in the scatter plot, showing that these queries have higher number of similar counterparts with respect to their pairs in the same bucket.

The entire logs could be scanned with input queries for the similarity search quite efficiently in less than 20 minutes without further optimizations.

[2]The same analysis can be applied to the other logs, but is probably less interesting for the present discussion.

[3]On August 23, 2018. Jena version same as in Section 2.1.

6

Software

The following sections focus on the software components that were developed during research to analyze and compile the results presented in this work.

As such, first a general outline is given, which is intended to aid in understanding the software, and highlight entry points that can be used for modifying and extending it. Then, the usage of the software and the output is discussed. This shows how all results can be reproduced and inspected, or how the software can be applied to different input data to perform the same analyses. Finally, the software is not only capable of performing analyses, it is also possible to explore query logs to pinpoint areas of interest and devise new analyses; as such this part of the software is presented and explained in more detail.

A goal of some summaries in the following is also to document the software from a very high-level perspective, so it can be understood and the source can be picked up with greater ease.

6.1 Architecture and Components

This section is intended to serve as a high-level overview of the software parts. The core component is written in Java, but there are also scripts written in Python, Groovy, and Bash, as well as external binaries, and embedded parts in other languages like of SQL.

The diagram in Figure 6.1 contains the most important parts of the program to get an understanding how the parts of the software are composed and interact. In the diagram, nodes surrounded with an ellipsis denote the actual class names as they are used in the Java part. Nodes in rectangular boxes are logical components that are used to group concepts. The upper part describes the flow of input, the middle part the output, and the bottom

parts the analysis. These parts can also be used independently for other purposes as well.

Starting from the top, the *Main* class is the program entry point that dispatches to a component of choice. This is also used to invoke utilities and tests, which is not shown in the diagram. For most purposes, this component simply invokes the *BatchProcessor*. This component can read and traverse directories or archives of files (e.g. compressed tarballs). It abstracts the concept of a filesystem with a virtual file system (VFS, see interface *Vfs*), so new archive formats can be added. But if enough file system storage is available, the usage of uncompressed log files is recommended for performance reasons. It is important to align the input files in a fashion that is useful for further analysis. For example, DBpedia logs were split into sources of origin, while Wikidata logs were split by the properties robotic and organic, and timeout and non-timeout. This split can simply be achieved by putting logs into different top-level directories. Each top-level directory is logically grouped then directed towards a concept that is called a *Job* (see interface of the same name) in the program.

The *BatchProcessor* basically tries to read all input contents with optimized buffers as then directs this input reading to the next step. The class *FileDispatcher* is responsible for a single log, since there are multiple log formats (CSV, CLF, etc.), this components dispatches the reading of input based on the format of the log to the next component. The class *FileHandler* is streaming the contents of the files, and hands of the processing to the appropriate log format parser. Query logs formats have two main types: In line type formats, a single line corresponds to a query and its metadata, and in multiline type formats, a single query can span multiple lines.

After passing these stages, a grouped set of queries is finally used as input for various jobs. Most jobs are classified as a *asyncJob*, which means that the steps of the jobs take the input queries that are fed into them, and queue them up asynchronously for parallel execution. But this does not work for all jobs. For example, the *StreakAnalysis* requires that all queries are run synchronously in order, as each query relies on the previous one to perform a temporal analysis. Finally, there are also some more complex jobs that do not fit into one of those two categories, as they have different configuration options that they can be run either synchronously or asynchronously in other parts. This mainly concerns

- the *Deduplicator*, which is used solely to produce unique query logs, which then can be used as new input for analysis jobs,
- and the *DatabaseFiller*, which is used to put queries and results of the analysis into a relational database for exploration and further analysis.

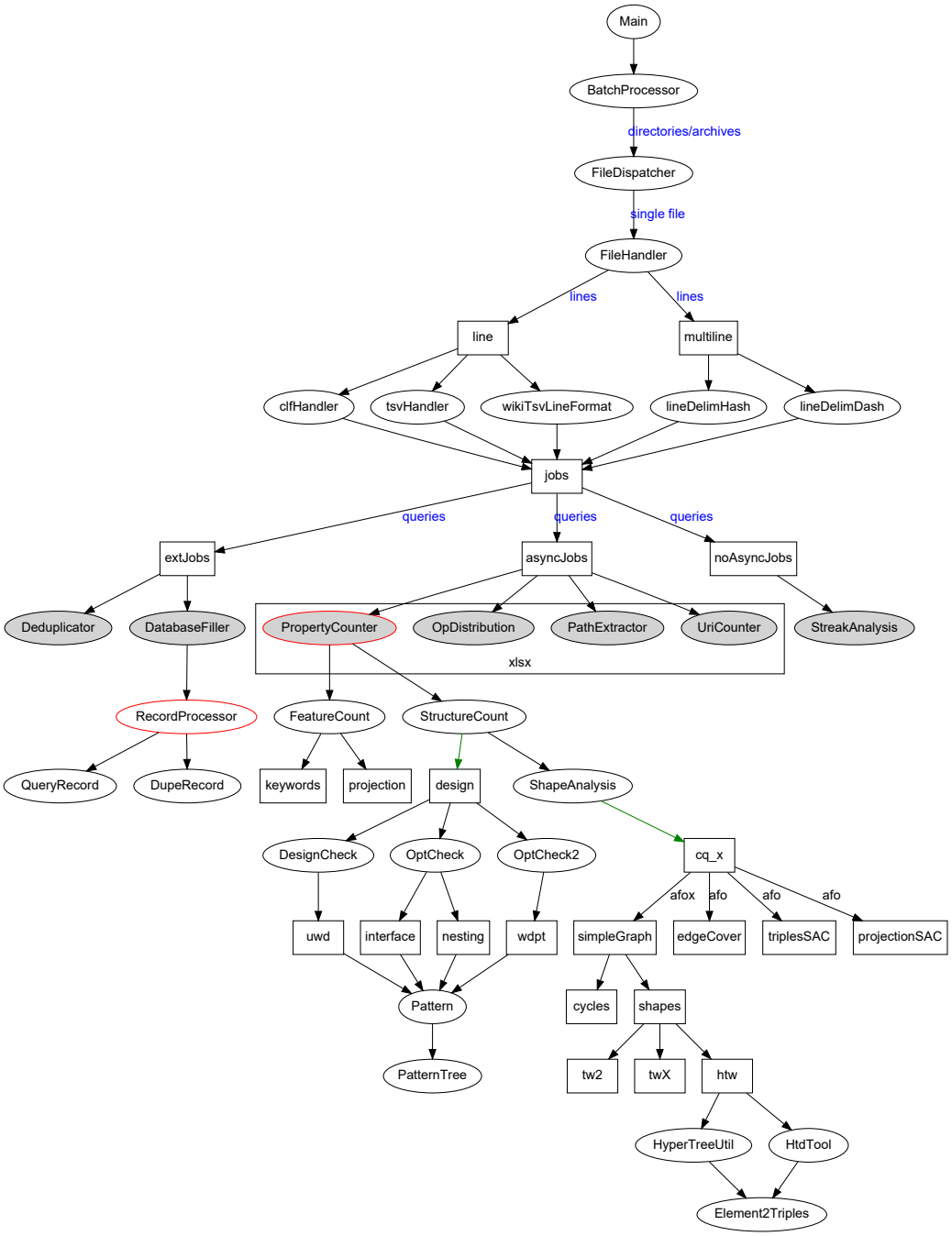


Figure 6.1: Overview of core components.

Most asynchronous jobs (surrounded by a box) are analysis jobs that use counters to store results of analyses for grouped sets of input files and output these results to YAML key-value files. A script transforms these files to XLSX spreadsheets, which can be directly used, or they can loaded into a tool like Jupyter for further analysis steps. These analysis jobs are

- *OpDistribution* to compile operator distributions for different operator combinations (see Section 2.3.2).
- *PathExtractor* to extract property paths from queries (as used for Chapter 3).
- *UriCounter* to create the sunburst diagram (Figure 3.1).
- *PropertyCounter*, the main component that contains counters for analysis properties.

The *PropertyCounter* and *RecordProcessor* used by the *DatabaseFiller* use a red ellipsis to denote that they are complex components that perform the core analyses. The *DatabaseFiller* also does deduplication and stores a *DupeRecord* for a duplicate query, and results of the analyses are stored in a *QueryRecord*. For *PropertyCounter*, the components for the analyses are shown under it:

- *FeatureCount* does keyword (Section 2.3.1) and projection (Section 2.3.4) analyses.
- *StructureCount* does analysis concerning query design (Section 4.1.3) and shape analysis (Section 4.2).

As can be seen, the class *ShapeAnalysis* is one of the most complex classes. It splits a query up into eligible fragments (Section 4.1.2) denoted by `cq_x` and runs applicable tests on them based on their graph or hypergraph.

6.2 Reproducibility of Results

The software is open-source and available at [Tim19a] on Github. It is licensed under the Apache Software License 2.0. There are full instruction on how to run build and run the software. There are scripts that can be used to do this, this will also try to download all tools including external tool binaries. These can be used to build Docker images to execute analyses on any platform.

All results in this work are reproducible by rerunning the analyses. The Wikidata Collection is publicly available and there is a script called `prepare_wikidata.sh` that can be used to download, decompress, and align the query logs for the analyses as performed. The Multi-Source Collection is not publicly available, but there is a script for preparing the files.

Finally, there are scripts for running a complete analysis or build a database from groups of input files. For Wikidata Collection, this script is called `batch_wikidata.sh`. By default, the output will be in a directory `xlsx` for sheets, and in a directory `batch` with subdirectories for each group of logs. The script for producing the spreadsheet is called `XLSX.groovy` and can be rerun on the output in `batch` to produce a spreadsheet.

A group of logs has the following output files:

- `property_count.yaml` is the main output for various general and one-off properties.
- `op_distribution.yaml` for the operator distribution.
- `pp_sparql.txt` contains SPARQL queries with property paths, separated by an extra empty line to denote a new query.
- `prop_paths.txt` contains only the property paths, used for compiling results in Chapter 3.
- `shapeless.txt` contains unclassified shapes from the shape analysis on CQs.
- `shapeless_f.txt` contains unclassified shapes from the previous analysis augmented by adding Filter.
- `shapeless_fo.txt` contains unclassified shapes from the previous analysis augmented by adding Optional.
- `shapeless_fov.txt` contains unclassified shapes from the previous analysis augmented by adding Values.

- `shapeless_fox.txt` contains unclassified shapes from the previous analysis augmented by adding Service and Bind.

Results from `prop_paths.txt` can be anonymized/normalized with the script `anon_prop_paths.py`, but this is also done by the batch script, which also produces a file `prop_path_count.tsv` that contains the count of property paths per group of logs as presented in Tables 3.2, 3.4, and 3.3.

A spreadsheet contains several sheets, they are produced from the YAML files discussed above, and a file `prop2cat.yaml` is used to control how properties are grouped to sheets. This can also be customized with scripting as done in the batch scripts. The sheets are:

- `property_count` with all remaining properties from `property_count.yaml` that are not moved by the scripts.
- `keywords` containing the count of queries that use given keyword at least once. May include operators that do not have a real explicit keyword such as And or Subquery. Results are presented in Section 2.3.1.
- `op_distribution` contains a count of comma-separated sorted operator combinations that are present in queries. Results are presented in in Section 2.3.2.
- `tw`, `htw`, and `fca` for treewidth, hypertreewidth (Section 4.2.2), and free-connex acyclicity (Section 4.2.3) with suffixes `_f`, `_fo`, `_fov`, and `_fox` as discussed above (with output files). All subsequent properties also have variants with these suffixes.
- `shapes` combined with *prefix combinations* of `re_` for fragments including property paths, `nc_` for a variant without constants. All subsequent properties also have variants with these prefixes. Results are presented in Section 4.2.1
- `cl_max` and `cl_min` for maximum and minimum length of cycles in cyclic queries. Results are presented in Section 4.2.3.
- `tripleCount` and `tripleSymbolCount` for counting the number of triples or triples including distinct symbols in predicates. Results are presented in Section 2.3.3.
- `varCount` and `constCount` for count of variables and constants per query in Section 2.3.3.
- `treePattern` for the tree pattern results in Section 3.4 (also available with suffixes).

- For quantitative shape properties for shapes (Section 4.2.3):
  - Chain queries with prefix `chain_` followed by
    - `depth_max` for the longest path in the graph of a query.
  - Star queries with prefix `star_` followed by
    - `depth_max` like for chains.
    - `degree_max` for the highest degree of any node.
  - Tree queries with prefix `tree_` followed by
    - The same properties for chains and stars.
    - `inner_deg_avg` for the average degree of all inner nodes.
    - `inner_deg_max` for the highest degree of all inner nodes.
    - `inner_tot` for the counting the number of all inner nodes.
    - `inner_rel` for the number of inner nodes divided by the number of all nodes of a query graph.
    - `split_tot` for the number of nodes that have a degree higher than two.

The classes identified in `shapes` (available in variants with prefixes and suffixes) for Section 4.2.1 are:

- `selfLoops` for queries containing self-loops.
- `parallelEdges` for queries containing parallel edges.
- `noNode` for queries with an empty graph.
- `noEdge` for queries that at most contain nodes.
- `singleNode` for queries with only a single node.
- `singleEdge` for queries with only a single edge.
- `singleEdgeSet` for queries with only disconnected edges.
- `chain` for chain queries.
- `chainSet` for sets of chain queries.
- `star` for star queries.
- `tree` for tree queries.
- `forest` for sets of tree queries.

- `flower` for flower queries as described in Definition 4.2.1.
- `flowerSet` for bouquets.

The results for projection in Section 2.3.4 use the following properties (available with suffix variants):

- `projection_cq` for Select queries positively identified using projection.
- `projectionUnsure_cq` for Select queries that use constructs that did not allow for positive identification of projection.
- `askProjection_cq` for Ask queries positively identified using projection.
- `askProjectionUnsure_cq` for Ask queries that use constructs that did not allow for positive identification of projection.

For the well-designed results in Section 4.1.3, properties are:

- `wd` for well-designed queries.
- `wwd` for weakly well-designed queries.
- `uwd` for unions of well-designed queries.
- `uwwd` for unions of weakly well-designed queries.

Important properties for reconstructing results with total numbers (for example for Section 2.1) are:

- `total_queries` for number of queries read.
- `total_valid` for number of parsed queries.
- `count_cq` for each shape variant (with prefixes/suffixes) as discussed above.

Note that these names are also mapped to columns names in the database. And the strings can be used to find search for the constant definition in the source code code to trace back to the implementation of tests.

6.3 A System for Exploration: DARQL

A plethora of SPARQL endpoints[1] is proliferating on the Internet thus allowing ordinary users to specify their queries either via APIs or manually. But understanding and exploring these vast amounts of data is challenging. The queries are collected into log files by their respective owners and represent a valuable resource for understanding users' preferences and needs in terms of query specification, but also for guiding us in research on query language design, query evaluation and benchmarking [BMT17a]; [KK16].

To help facilitating the research of SPARQL logs, we developed DARQL, a tool for deep and fast analysis of large SPARQL query logs. The tool comes equipped with an extensive set of pre-defined tests, including simple tasks (keyword counts, triple counts, operator distributions), moderately deep tasks (projection test, query classification), and deep analysis (shape analysis, well-designedness, weakly well-designedness, hypertreewidth, and fractional edge cover). The primary goal of our tool is to let the users dive into SPARQL query logs and let them discover the inherent characteristics of the queries. DARQL is an easy-to-use tool for SPARQL query analysis in the research community. Out of the box, DARQL analyzes 62 properties per query. We believe that DARQL will give researchers who want to dive into query log analysis a significant head start. Indeed, in our former analytical study [BMT17a] we only scratched the surface when it comes to finding correlations between query properties.

There are publicly available logs that can be used [Bie+18] that can be used, or other non-public sources that can be obtained as outlined in [BMT17a]. We release the tool itself at `https://github.com/PoDMr/darql`.

6.3.1 System and Main Components

A query builder lets the user modify the features of the queries under scrutiny to respectively enlarge or restrict the scope of the analyzed portion of the corpus. Since our tool is deployed on top of a relational DBMS with a web-based front end, each search on the corpus corresponds to an SQL query issued on the database. The user can also manually modify this query and rerun a deeper or coarser analysis at will.

Internally, the system consists of the following main components:

- a batch processing system (for loading and analyzing query logs, writing to files),

[1] `https://www.w3.org/wiki/SparqlEndpoints`

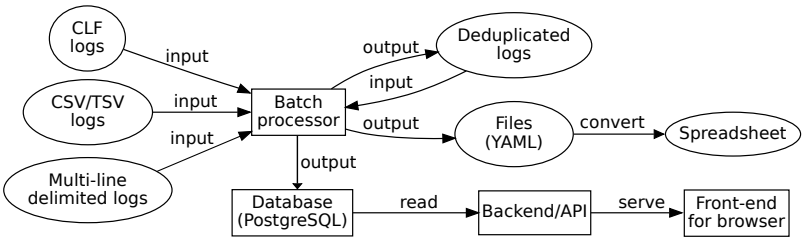


Figure 6.2: General architecture of the DARQL system.

- a PostgreSQL 10 database, and
- a GUI served from a connecting back-end.

Refer to Figure 6.2 for a more detailed overview. The main components are described next.

Loading and Analyzing Query Logs We release DARQL as open-source project and therefore also discuss some aspects that surround it, such as the batch processing system. Before the user can start exploring queries through the database interface, the queries need to be analyzed, deduplicated, and stored into the system. To this end, we provide scripts that can handle three main log formats: CLF-based logs (Common Log Format) used by most web servers, delimiter-based line log formats (e.g. CSV, TSV), or multi-line delimited formats. Every query in the log is parsed and stored into the database. For queries that do not parse, we record this in the database and do not perform further analysis. Every query that parses is run through an extensive set of analytical tests: 1 parse test, 31 keyword tests (query type, operators, solution modifiers, aggregation operators, ...), 8 simple structural tests (property path, projection, ...), 4 well-designedness tests, 3 classifications into different kinds of conjunctive queries, 11 complex shape tests for the structure of conjunctive queries (chain, star, cycle, tree, flower,...), and 4 value tests (number of triples of the queries, hypertreewidth, fractional edge cover, and the origin of the query logs). These tests include (but are not limited to) all those that have been used in [BMT17a].

Prior to analysis, we test for duplicates in the query logs. We use SHA-256 for hashing to detect potential duplicates. The strings of queries are normalized by outputting them with the Jena parser, which normalizes whitespace and formats the queries in a readable form for our GUI. For some of the logs, we also need to add implicit prefixes explicitly to make queries valid as

standalone queries. If we discover duplicate queries we do not re-run query analysis, but simply record its occurrence by referring to the first occurrence and store the new origin (log file and line number). As such, our system can display, for each query, how many duplicates were found and where.

The batch processing system can write output to either files or databases. It also allows logs to be rewritten in different formats, and deduplicated in a normalized form. The analysis output is in machine readable formats, and it can be transformed to spreadsheets.

Database The database is a PostgreSQL 10 system that stores the results of each analysis for every query. For duplicate log entries it stores the origin of each entry. We used PostgreSQL 10 in order to utilize new parallelization features for queries and joins, which after appropriate tuning resulted in much faster query times compared to PostgreSQL 9.6.

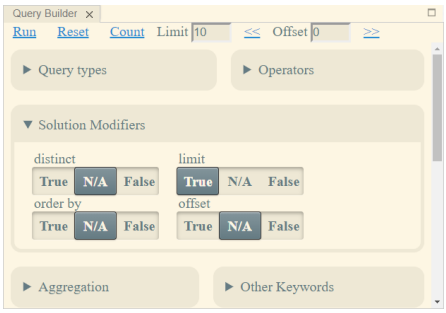
We tested the system through a web interface on our server[2]and noticed that most queries that our Query Builder generates are typically done in less than a second (e.g. 200ms). Count queries are generally more expensive in PostgreSQL and take between 2–4 seconds on large sets (50–100 million queries) and are faster if subsets are smaller. In order to achieve this performance, we created a set of indexes. Depending on the queries, one can try to get faster queries, by creating more specific indices, but they can get very large. One also has to consider that the indexes a specific user really needs can vary very widely. Depending on data, index creation can take up to 2–4 minutes.

All non-duplicate queries are stored in a single table `Queries` with the information of analysis results as Boolean or numeric columns. Each query is assigned a unique ID. Additionally, it contains the string of the query, a hash of this string, and its origin (data set, filename, line number in file). Duplicates are stored in a table `Duplicates`, containing an ID for the duplicate, a reference ID to the original query, and the origin of the duplicate.

As an alternative, the analysis results could have been stored in an additional separate table, but this approach was not chosen, because the queries from the interface often fetch all analysis results anyway, since this is much more efficient for the interface and the user. However, from a software architectural standpoint, it is possible to easily change this in the code, the queries for the interface would then need to be done with more joins.

The schemas are generated by the batch processor if the database is empty. It is possible to do this manually, which allows to use a different

[2]A 2-CPU Intel Xeon E5-2630v2 2.6 GHz server with 128GB RAM and running Ubuntu 16.04 LTS.



(a) Screenshot of the Query builder, showing a few of the properties (with "Solution Modifiers" unfolded)

(b) Automatically generated SQL query (this one fetches the SPARQL select-queries with hypertreewidth two)

Figure 6.3: Query Builder and SQL text editor.

order of columns if desired, and it allows changes to the schema so columns for new tests could be added or data could be stored in additional tables that could be joined as needed.

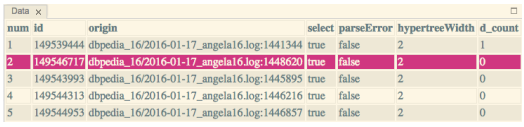
6.3.2 User Interface

The user interface consists of several components that are connected. The main ones are: (1) a query builder, (2) an SQL text editor, (3) a query visualizer, (4) a SPARQL text display, and (5) a query result table display.

In a typical usage scenario of the tool, the user employs the query builder to select properties that she is interested in. Figure 6.3(a) shows a partial screenshot of the query builder. Under "Query Types", one can click if one wants to search for Select, Ask, Construct, or Describe queries. Likewise, we have categories for "Operators", "Solution Modifiers", "Well Designedness", "Shapes", etc. As Figure 6.3(a) shows, groups of properties can be folded and unfolded at need. Boolean properties can be set to True, False, or N/A (don't care). For numerical properties (e.g., number of triples, hypertreewidth), the user enters can enter a number or a range. A range is specified by using a comma to separate lower and upper bound, either one can also be omitted.

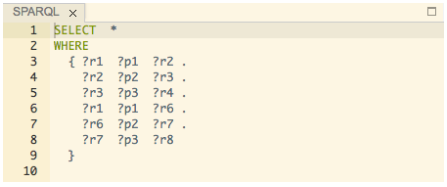
The GUI automatically generates an SQL query interactively when properties are changed in the query builder, see Figure 6.3(b) for an example query. The SQL query in the editor is fully editable, in case the user wants to refine the search. It can be executed by several means, the most obvious way is clicking "Run". By default, a query execution fetches up to ten queries from our query logs and displays the first retrieved query as current query.

Upon executing the SQL query, several things happen at once:



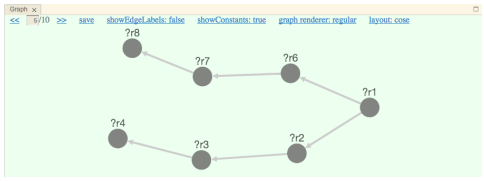
num	id	origin	select	parseError	hypertreeWidth	d_count
1	149539444	dbpedia_16/2016-01-17_angela16.log:1441344	true	false	2	1
2	149546717	dbpedia_16/2016-01-17_angela16.log:1448620	true	false	2	0
3	149543993	dbpedia_16/2016-01-17_angela16.log:1445895	true	false	2	0
4	149544313	dbpedia_16/2016-01-17_angela16.log:1446216	true	false	2	0
5	149544953	dbpedia_16/2016-01-17_angela16.log:1446857	true	false	2	0

(a) Query result table (second entry is highlighted).

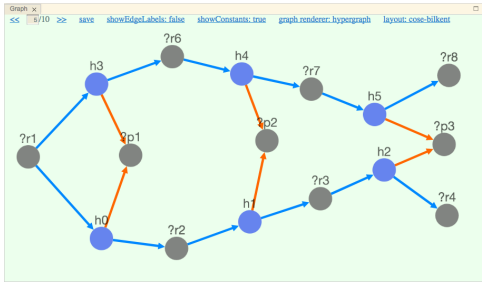


```
1  SELECT  *
2  WHERE
3    { ?r1  ?p1  ?r2 .
4      ?r2  ?p2  ?r3 .
5      ?r3  ?p3  ?r4 .
6      ?r1  ?p1  ?r6 .
7      ?r6  ?p2  ?r7 .
8      ?r7  ?p3  ?r8
9    }
10
```

(b) One of the queries in our log (in the SPARQL editor).



(c) Visualization of the graph of Figure 6.4(b)'s query.



(d) Visualization of the hypergraph of Figure 6.4(b)'s query.

Figure 6.4: Partial screenshots of the query viewer and visualizer for an example query as regular graphs.

- its results are shown in the query result table display (shown in Figure 6.4(a));
- the first result is displayed in the SPARQL text display (shown in Figure 6.4(b)) and
- the first result is visualized in the query visualizer (shown in Figure 6.4(c) and 6.4(d)).

The entries in the result table display are clickable, so the user can immediately select a query she is interested in (e.g., the second entry in Figure 6.4(a)). When clicking a query, it is shown in the SPARQL editor and visualized (e.g., Figure 6.4(b) –6.4(d)) show the highlighted query from Figure 6.4(a) and two different visualizations. Furthermore, the query visualizer has controls for going to the previous and next query. Additional properties of the currently displayed query can be shown in a "Details" panel.

Figure 6.7(c) shows a query from the DBpedia 2016 (Jan. 17th) log file, corresponding to one of the results of the SQL query in Figure 6.3(b) on our database, fetching the queries of hypertreewidth two. The visualizer automatically renders a visualization of the query and can be configured to render the graph- and hypergraph structure. Although the graph structure

of a query is usually rather simple, it is often not sufficient to convey the full complexity of the query. For instance, the graph in Figure 6.4(c) ignores the variables ?p1, ?p2, and ?p3 from Figure 6.7(c).

The hypergraph structure captures this complexity more accurately. Since we only had graph render libraries at our disposal, we display a hyperedge $\{s, p, o\}$ coming from SPARQL triple (s, p, o) as a new node h (representing the identity of the hyperedge) which we connect to nodes s, p, and o. We use the edges $s \to h$ and $h \to o$ (in blue) and $h \to p$ (in orange). Figure 6.4(d) has such a visualizations for the SPARQL query in Figure 6.7(c).

The visualizer currently uses several graph layout algorithms (cose, cose-bilkent, concentric, breadth-first, grid, and circle) and can readily switch between them. This gives users a quick idea of the query's structure.

Note that the SPARQL editor can be used to edit the current query, and the SPARQL visualizer is immediately updated to show changes. The SPARQL visualizer also allows manual layouting by dragging nodes, and it can also be zoomed and moved around.

Finally, we provide a data sets panel (shown in Figure 6.5), which shows general statistics of the data that is currently in the database, ordered by origin. The panel contains four columns: name of the data set ("originMajor"), total number of queries ("total"), number of unique queries ("unique"), and finally the number of unique queries that can be parsed ("unique_valid"). The total number of queries in each log is the sum of the "dupe" and "nodupe "values. Notice that some logs contain more duplicate entries, while others contain more unique entries. Duplicates are tested globally, so we may classify a query from dbpedia_15 as a duplicate if it already occurred in dbpedia_12.

In the GUI, it is possible to individually move, resize, or maximize all panels, this allows to give the user a complete overview. It also allows to hide the panels the user is currently not interested in. The panels can be used as stacked tabs or as split views with advanced layout capabilities found in full-fledged IDEs.

6.3.3 Usage Scenarios

DARQL is a user-friendly system to explore query logs, as such, it is interesting to see how queries from actual SPARQL log files (DBpedia, BioPortal, LGD, OpenBioMed, Semantic Web Dog Food, British Museum; Wikidata, or any other logs one decides to feed into it) look like. Since DARQL is very flexible, and immediately shows visualization such as in Figures 6.7(b) and 6.4(d), this can be done interactively and with ease. . Nevertheless, there

Data sets ×

originMajor	total	unique	unique_valid
dbpedia_12	28,651,075	14,086,448	13,612,284
dbpedia_13	5,243,853	2,731,415	2,422,257
dbpedia_14	37,219,788	18,265,430	16,407,765
dbpedia_15	43,478,986	13,098,872	12,768,271
dbpedia_16	15,098,176	3,941,745	3,810,806
lgd_uw13	1,927,695	639,090	364,077
lgd_uw14	1,999,961	668,807	636,963
bioportal_uw13	4,627,270	688,523	688,142
bioportal_uw14	26,438,932	1,534,577	1,525,274
biomed_uw13	883,375	832,779	832,268
swdf_uw13	13,853,604	1,408,945	1,355,638
RKBE_tsv	1,555,940	135,178	135,122
wikidata	309	308	307

Figure 6.5: The data sets panel from the GUI, showing the currently loaded queries.

are specific scenarios that come to mind how the system can be used, which will be discussed in the following.

Search for Complex Queries Besides searching simple queries in our large corpus, the tool allows us to quickly search for queries by size (so the largest ones can be found quickly), with complex structures (e.g., cyclic queries), and with advanced keywords. Since our corpus encompasses a varied set of SPARQL log files coming from disparate SPARQL endpoints, DARQL lets the users access the lineage of the queries under inspection and have a perception of what logs contain queries with certain complex characteristics.

Shape-Driven Exploration Here we start by selecting a specific shape (e.g., "star") and show visitors on the visualizer how star-shaped queries in the log files actually look like. This gives an impression on the size, complexity, branching, and diameter of such queries. The tool supports many different shapes to start from, such as star, tree, chain, forest, flower, and cycle. We can also start from a given hypertreewidth, which might be desirable, since queries with hypertreewidth three are already quite complex and rare in practice (as seen in in Section 4.2.2).

In fact we already used the front-end extensively for our study of shapes in [BMT17a]: we gradually implemented more and more shape tests and then visually inspected queries that were not classified by any known shapes. So, this part of the tool has already been heavily used behind the scenes in our

own research. It helped us identify new shapes and we could gradually cover almost all queries of a limited treewidth with a defined shape.

Getting Statistics DARQL can get statistics such as "How many of the SELECT queries are conjunctive queries?" or "How many of the construct-queries do a non-trivial insertion?" and it is possible to immediately display these queries. For every query that can be constructed with the query builder, we can toggle if the tool should count the number of the results, or if it should produce a sample of the answers. Therefore, such statistics can be easily computed.

Most Popular Queries Statistics on the logs can be easily computed and lead to identify for instance the most (or the least) occurring queries in the entire corpus or in a single log file or data source. We can thus access the most (or less) popular queries in the logs with a breakdown view on each individual log file, on each individual data source (e.g., Wikidata, DBPedia, BioPortal etc.), or on the entire corpus.

Expert Search in SQL The predefined user interface only generated SQL queries to the database that test conjunctions of conditions. Using the direct SQL interface, we show that if the user wants, also more advanced conditions can be queried. For example, one can search the union of all queries that have a minimum size and those that have cycles.

Furthermore, it is possible to explore queries in the context of time. Although only some logs have timestamps, most log files have names that indicate a date. Figure 6.4(a) shows five queries from January 17th, 2016, for example. Using the SQL editor, queries coming from a specific date (or month, or year) can be found as well. With this date, we could perform complex interesting queries with a temporal aspect. For instance, we could inspect how many queries were submitted on the same day, or try to find days or time spans that have the most or least queries. Or, we could calculate the time span (first and last occurrence) for duplicates of a query. User can design their own very complex log searches by formulating them as queries in SQL.

6.4 Advanced Visualization and Statistics: SHARQL

SHARQL builds on DARQL [BMT18], a system deployed on top of a relational DBMS (PostgreSQL 10) with a web-based front end alongside a batch processing system (for loading and analyzing query logs, writing to files) and initially conceived for DBpedia queries. DARQL [BMT17a]; [BMT18] was however working on an initial batch of DBpedia logs augmented with logs from other sources and roughly amounting to 1/4 of our current huge corpus in SHARQL (whose data set breakdown is illustrated later in Figure 6.11). Since the logs from [Bie+18] is publicly available, there is a script to import them automatically into SHARQL. Due to the fact that additional large query logs have been added since then, the new system SHARQL has an array of new features including:

- Adaptive edge rendering suitable for hypergraph visualization of queries; coupled with tree decomposition visualization.
- Precise shape analysis (with/out constants) and property paths support in visualization and shape analysis.
- A much larger query fragment wrt shape analysis covering C2RPQs (Conjunctive Two-ways Regular Path Queries). The latter fragment is 57x times larger than the fragment of recursive queries found in previous logs (including DBpedia).
- Precise analytics on a rich variety of query features, such as spanning size, diameter, cycle lengths of queries alongside the numbers of nodes with branching, maximum node degree, edge cover, and total sum of triples and symbols in property paths. These analytics can be obtained for each particular class of queries exhibiting one of the discovered query shapes as well as for organic ("human-written") and robotic ("automatically generated") queries, and for timeout and well-executed queries.

The code base for the query analysis and visualization is available at `https://github.com/PoDMr/sharql`. An additional advantage of SHARQL is that the 200M Wikidata queries are freely available for download [Mal+18] and can be imported into the database using our scripts.

Advanced Usage Scenarios SHARQL allows to visualize the queries of SPARQL log files with rather different characteristics (Wikidata, DBpedia, BioPortal, LGD, OpenBioMed, Semantic Web Dog Food, British Museum),

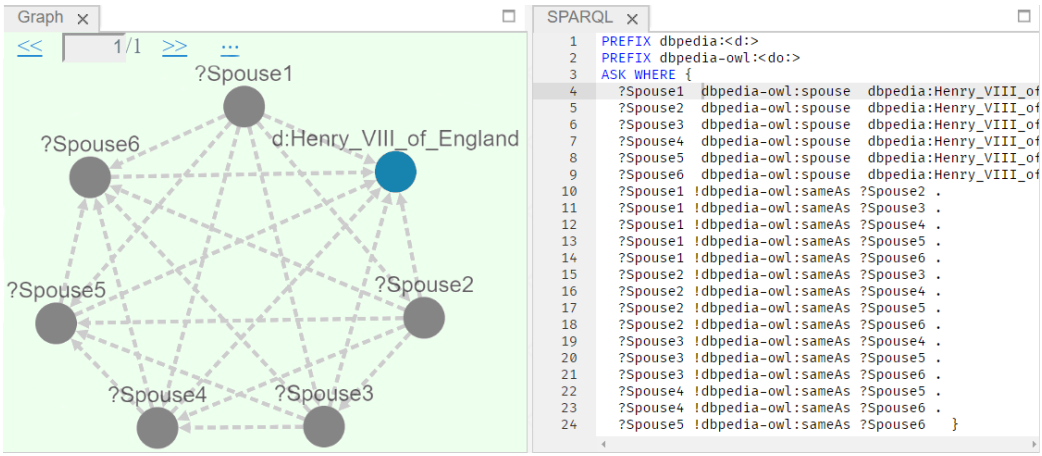


Figure 6.6: The Henry VIIIth Query.

out of which there are recently publicly available logs and undisclosed logs.[3] Furthermore, the system is able to digest a huge amount of queries, currently about 500M in total. To illustrate the system more, here are some scenarios for using SHARQL.

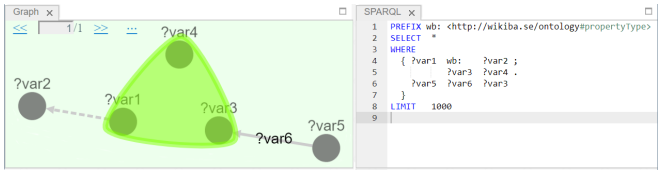
6.4.1 Rendering and Visualization of Queries

In order for a human to quickly understand queries, visualization may be very helpful. For instance, consider the query in Figure 6.6, coming from DBpedia. The query consists of 21 edges involving one constant ("Henry VIII") and six variables. Whereas the query itself (Figure 6.6 right) takes some time to parse, a visualization immediately shows that its shape is a 7-clique and that the user may be interested in obtaining the six spouses of Henry VIII.

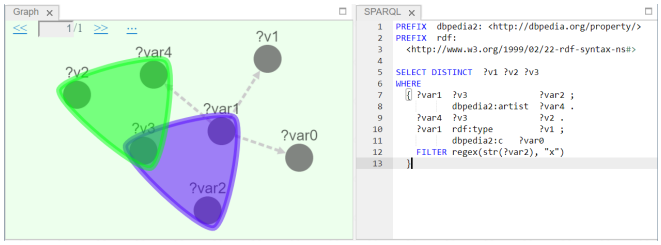
Furthermore, the graph- or hypergraph structure of queries gives crucial information concerning the complexity of evaluating them. In the case of case of *conjunctive queries*, it is well known that acyclic queries can be evaluated efficiently, whereas queries that are highly cyclic are very complex. (For instance, if a query's shape is a k-clique, then evaluating it is equivalent to solving the NP-complete k-clique problem.)

Central aspects to query visualization are the rendering of *constants* and *variables*, *property paths*, and *hypergraph representations*. We will prepare a set of queries to show visitors how these aspects are visualized.

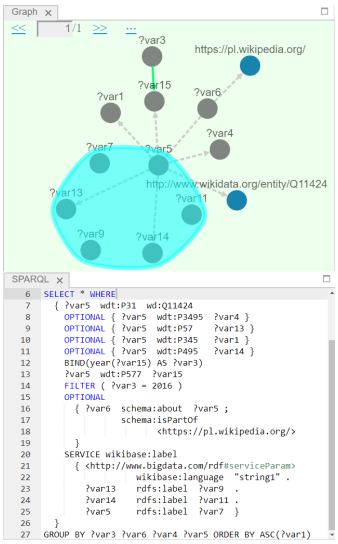
[3]The massive DBpedia logs are not publicly available and have been given to us courtesy of OpenLink Software.



(a) An organic query from our logs featuring all kinds of edges



(b) A complex (organic) query from our logs prone to hypertree decomposition



(c) A complex organic query from our logs

Figure 6.7: Partial screenshots of our query viewer and visualizer for example queries with hypergraph components.

Hypergraph rendering of queries and navigation In the visualization of SPARQL queries, the necessity for hyperedges (edges with more than two nodes) arises when rendering queries that use complex subqueries using the Values, Service, Bind and Filter operators from SPARQL. Hyperedges may also necessary when variables are used in the *predicate* position of graph pattern triples, but we noticed that in practice this is needed much less often than with the abovementioned keywords. We therefore implemented a hypergraph rendering algorithm, which we will show in action here. Figure 6.7 contains three example queries and their rendering as hypergraph.

We let the user navigate through our massive logs and visualize the hypergraph rendering of queries, starting with a few selected queries like the ones in Figure 6.7. Query log exploration can then be done in many different directions: we ran about 120 tests on every query in the logs. Any of these tests can be used for further exploration. For instance, users can search for queries with the largest hyperedge in the logs; focus on queries within the subsets of bot queries or user queries; focus on timeout queries, etc. Later, we will explain our novel hypergraph rendering of queries and the rendering of Fig. 6.7 in detail. We note that all our graph layout algorithms are interactive: the user can click a node and drag it, and the layout changes dynamically.

Property Path Visualization and Related Types Property paths are extremely common in Wikidata query logs [BMT19b]. However, they can consist of complex expressions and can therefore quickly overcrowd the visualization. We solved this by rendering property paths as special (dashed) edges, without any annotation, for which the property path expression can be seen by hovering over the edge with the mouse.

Impact of Constants on Query Shapes We want to show how the shape of queries changes if constants are removed from their (hyper-)graph. We noticed that the shape of many queries becomes *disconnected* if constants are removed [BMT19b]. This means that we can show on actual examples that it makes much sense for query evaluation to start with searching the constants in the query in the data and expand to variables later. This avoids the computation of huge Cartesian products (joins) that are only pruned later when the constants are added.

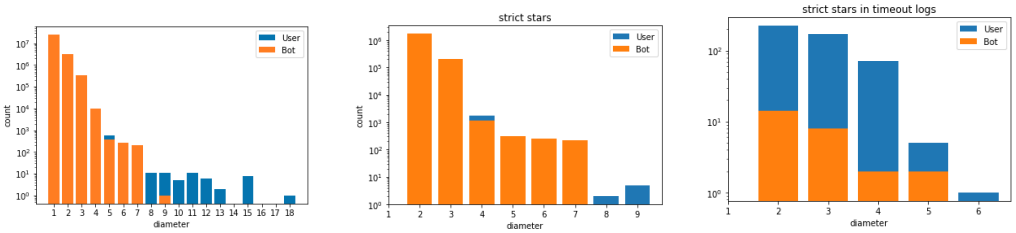
6.4.2 Flexible Analytics

We have subjected each query in our database to a total of roughly 120 tests, compared to 62 we did in DARQL [BMT18]. These tests involve information on the set of operators it uses, its number of triples, its graph shape, its hypergraph shape, well-designedness, weak well-designedness, acyclicity, free-connex acyclicity, whether it uses property paths, number of variables, number of constants, etc.

For queries that can be rendered as a graph, we test if it has self-loops (self-joins), parallel edges, we measure the diameter of its graph (longest distance between nodes), maximum degree, its number of nodes with high degree, length of its shortest and longest simple cycles.[4] Furthermore, our database records, if available, whether the query was *robotic* or *organic* and whether it timed out or not. SHARQL allows to use each of these analysis results to study subgroups of queries in detail. Under the hood, each query is stored together with 120 attributes in our query database, each of which represents the outcome of one of our tests.

Analytics on the Entire Logs and Subsets Thereof We can use our database as a back-end for Jupyter, which means that, for each of the individual analyses we did, bar charts or other charts (scatter plots, etc.) can

[4]Only up to length 10, since deciding if a graph has a long simple cycle is NP-complete. It is at least as difficult as the Hamilton Cycle problem.



(a) Diameter distribution (user vs. bot)

(b) Diameter distribution (user vs. bot) for star queries only

(c) Diameter distribution (user vs. bot) for star timeout queries only

Figure 6.8: Examples of diameter distributions on the entire Wikidata logs and subsets thereof.

readily be produced. Already these simple distributions show interesting insights on global properties of the logs.

But the integration with Jupyter is much more flexible and we can combine statistics in an almost arbitrary manner. We discuss one scenario as an example. Figure 6.8(a) shows a bar chart, obtained using SHARQL, with the distribution of the diameter of the graphs of user versus bot queries in Wikidata, roughly 208M queries. (Notice the log scale on the vertical axis.) A user may be interested in refining this result to, say, star-shaped queries only, for which we see the result in Figure 6.8(b). (Here, "star" is a predefined shape we identified in [BMT19b] and which is very prominent in the logs.)

The statistic can be refined even more by, e.g., only focusing on *timeout* star queries, see Figure 6.8(c). We can observe from the three views that user queries have significantly larger diameters than their robotic counterparts (Figure 6.8(a)). This is confirmed by star queries (Figure 6.8(b)), whereas for timeout queries the percentages are more balanced (Figure 6.8(c)). As an example, we give the Jupyter code we used to produce Figure 6.8(c) in Figure 6.9. When we computed these results, the system is quite responsive and answered (produced the output bar charts) within a matter of seconds.

The combination of the SHARQL database and Jupyter is extremely flexible, and gives the user more than 0.5B queries to analyze. The currently loaded data sets in the system are depicted in Figure 6.11.

Search for Complex Queries Besides searching for simple queries in our large corpus, the tool allows us to quickly search for queries by size (so the largest ones can be found quickly), with complex structures (e.g., cyclic queries), and with advanced keywords. Since our corpus encompasses

```
qu3 = (
    'SELECT "depth_max", count(id) FROM "Queries"',
    "WHERE \"originMajor\" = 'wd_user_500'",
    ' AND star = true AND chain = false ',
    'GROUP BY "depth_max"'
)
qb3 = (
    'SELECT "depth_max", count(id) FROM "Queries"',
    "WHERE \"originMajor\" = 'wd_bot_500'",
    ' AND star = true AND chain = false ',
    'GROUP BY "depth_max"'
)
ru3 = query(qu3)
rb3 = query(qb3)
```

```
barchart(ru3, rb3, 'strict stars in timeout logs')
```

Figure 6.9: Jupyter code used to produce Figure 6.8(c).

a varied set of SPARQL log files coming from disparate SPARQL endpoints, SHARQL lets the users access the lineage of the queries under inspection and have a perception of what logs contain queries with certain complex characteristics.

Hypertree Decomposition and its Visualization A *hypertree decomposition* of a hypergraph (similar to a tree decomposition of a graph) is a suitable clustering of its hyperedges yielding a tree or a forest. Such decompositions are important for database queries, since they can serve as a guide for join orderings. In SHARQL we can derive the hypertree decomposition of all queries in the logs that are sufficiently close to a *conjunctive query* so that hypertree decompositions make sense.[5] These queries amount to roughly 177.5M queries in the Wikidata data set only (and 351.3M for the entire data set). As an example, Figure 6.10 shows a hypertree decomposition of the query illustrated in Figure 6.7(c) with overlapping hyperedges.

6.4.3 Graph Rendering

In this section, we explain some visualization aspects in more detail. Our query visualizer can render the edges of a query as regular edges or hyper-

[5]We use detkdecomp (`https://github.com/daajoe/detkdecomp`) for obtaining the hypertree decompositions.

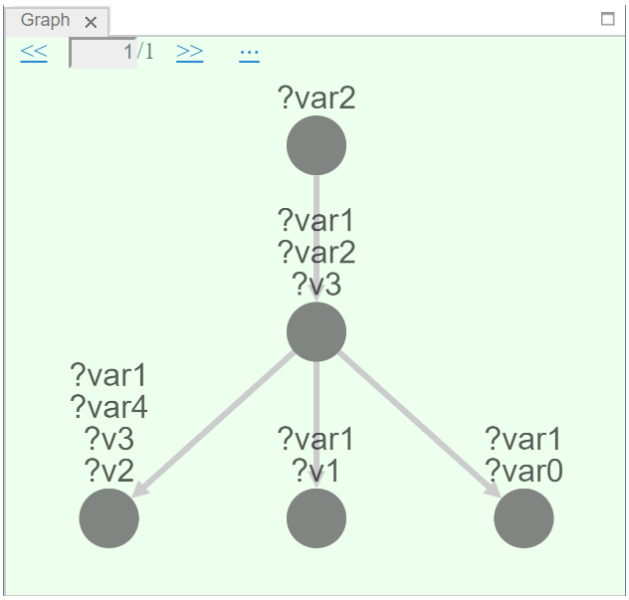


Figure 6.10: Hypertree decomposition of the query in Figure 6.7(b).

edges. Regular edges have two nodes (and possibly an edge label) and are either dashed or solid. A dashed regular edge is an edge for which the edge label is hidden, and for which the label appears when hovering over it. This is especially helpful for property paths, since these expressions are arbitrarily complex. Solid regular edges may be grey or colored. If it is colored (as the green edge in Figure 6.7(b)), it means that it is generated from a Service, Bind, Values, or Filter clause. (The green edge in Figure. 6.7(c) is generated from the Bind clause of the query.)

Hyperedges are edges that have more than two nodes. They can be generated in two scenarios. The first scenario occurs with a Service, Bind, Values, or Filter clause, which creates a constraint (and thus a hyperedge) between three or more variables or constants, as illustrated in Figures 6.7(c) for the blue edge, and 6.7(b) for the green and purple edges. The second scenario is illustrated in Figure 6.7(a). Here, the green hyperedge arises from an ordinary triple pattern $\langle s\ p\ o \rangle$, for which the p position ($?var3$), is a variable that is used elsewhere. Notice that we can deal with the triple pattern $\langle ?var5\ ?var6\ ?var3 \rangle$ differently. Since $?var6$ is not used elsewhere, we can render it as a regular labeled edge.

As shown in the screenshots, we also render variables and constants in a different color. This is crucial for being able to judge the complexity of evaluating a query, because a constant can only match to one node in the graph, whereas a variable can potentially match to any node. Moreover,

Data sets ×

originMajor	total	valid	unique_valid
wikibot	207,505,296	207,464,954	34,523,883
wikiuser	661,769	651,385	251,994
wikibot_timeout	33,616	33,465	3,168
wikiuser_timeout	14,528	14,087	8,729
dbpedia_12-17	298,801,919	288,173,920	85,347,571
lgd_13-14	3,927,656	3,483,137	986,483
bioportal_13-14	31,066,202	31,029,165	2,878,924
biomed_13	883,375	882,847	27,030
swdf_13	13,853,604	13,670,550	1,229,759
britm	1,555,940	1,545,643	135,112

Figure 6.11: The data sets currently loaded in the system.

constants play a crucial role in shape classification of our queries, since discarding them might lead to breaking up query shapes that consider both variables and constants [BMT19b].

In drawing queries as hypergraphs, we have used the notion of planarity introduced in [SO87]. Not all hypergraphs have a vertex-planar representation, but in the majority of the cases in our queries this is the case. (In general, determining if a given hypergraph has a vertex-planar representation is NP-complete [SO87].) We adapted the layout algorithm of Arafat and Bressan [AB17] in order to handle the visualization of ordinary edges and hyperedges in our corpus.

7

Conclusion

We will now reflect on the results and discuss them, see what lessons were learned, and what this means for future work.

7.1 Reflecting on Multi-Source Collection

An extensive analytical study on a large corpus of real SPARQL query logs was conducted. The Multi-Source Collection corpus is inherently heterogeneous and consists of a majority of DBpedia query logs along with query logs on biological data sets (namely BioPortal and BioMed data sets), geological data sets (LGD), bibliographic data (SWDF), and query logs from a museum's SPARQL endpoint (British Museum). This corpus was augmented with the example queries from Wikidata (Feb. 2017), which are cherry picked from real SPARQL queries on this data source.

Differences in Data Sets The majority of the data sets exhibit similar characteristics, such as for instance the simplicity of queries amounting to 1 or 2 triples. The only exception occurs with British Museum and Wikidata data sets, where the former is a set of queries generated from fixed templates and the latter is a query *wiki* rather than a query log. Clearly, the DBpedia data sets are the most voluminous and recent in this corpus, thus making their results quite significant. For instance, despite the fact that single triple queries are numerous in these data sets, more complex queries (with 11 triples or more) have lots of occurrences (up to 21% of the total number of queries for DBpedia13). Strikingly, the largest queries of all belong to DBpedia, which is one of the outcome of the comparison between *Valid* and *Unique* queries.

Moreover, it can be observed that most of the analyzed queries across all

data sets are Select/Ask queries, which range between 91% and 99.88% for all data sets except DBpedia16 and LGD13, which have lower percentages. Therefore these queries were put into focus in the analysis, since these queries turn out to be the queries that users most often formulate in SPARQL query endpoints. The occurrences of operator distributions and the number of projections and subqueries were further examined. This analysis addresses a specific fragment, namely the And/Optional/Filter patterns (AOF patterns). For such patterns, the graph- and hypergraph structures were derived and this allowed to analyze the impact of the structure on query evaluation.

Benefits of Shape Analysis We synthetically reproduced the observed real chain and cycle query logs with a synthetic generator by building diverse workloads of Ask queries and measured their average runtime in two systems, Blazegraph, used by the Wikimedia foundation, and PostgreSQL. In both systems, the difference between average performances of such different query shapes are perceivable. We dug deeper in the shape analysis in order to classify these queries under general query shapes as canonical graphs and characterize their tree-likeness as hypergraphs. This shape analysis can serve the need of fostering the discussion on the design of new query languages for graph data [Bar16]; [Bon+18], as pursued for instance by the LDBC Graph Query Language Task Force [LDB]. It can also inspire the conception of novel query optimization techniques suited for these query shapes, along with tuning and benchmarking methods. For instance, we are not aware of existing benchmarks targeting flowers and flower sets. The analysis on property paths showed that these are not yet widely used in the entire corpus, even though they are numerous in the Wikidata corpus.

Benefits of Streak Analysis Finally, we performed a study on the way users specify their queries in SPARQL query logs, by identifying streaks of similar queries. This analysis is for instance crucial to understand query specification from real users and thus usability of databases, which is a hot research topic in our community [Jag+07]; [NJ11].

Extensibility Our analysis has been carried out with scripts in different languages, amounting to a total of roughly $9,000$ source lines of code (SLOC). These scripts are open-source [Tim18] and extensible to the new query logs that will be produced by users on SPARQL endpoints in the near future.

Future Work A preliminary investigation on our data set showed that a shape analysis that incorporates property paths (and therefore considering

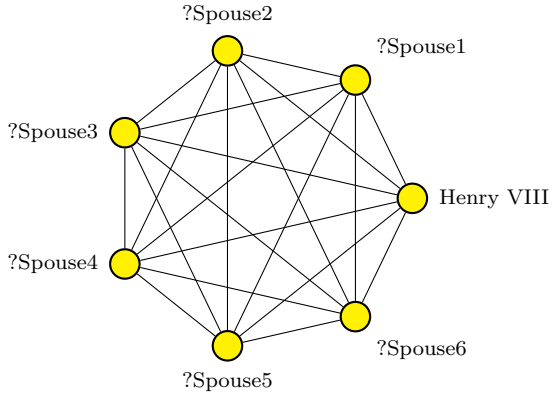


Figure 7.1: The Henry VIII query, a 7-clique containing one constant and six variables. All edges between Henry VIII and the variables are labeled "dbpedia-owl:spouse" and all edges between variables are labeled with the property path "!dbpedia-owl:sameAs".

extensions of CRPQs instead of CQs) may reveal interesting results. For instance, we found a 7-clique query (6-clique without constants) similar to the one in Figure 7.1. We also found this particular query interesting, because we believe that its semantics is probably different from what the user intended. We believe that the user wanted to search for (possibly all permutations of) six different spouses of Henry VIII. However, "!dbpedia-owl:sameAs" tests if there exists an edge between two nodes that is not classified as dbpedia-owl:sameAs.

7.2 Reflecting on Wikidata Collection

An in-depth analysis of the recently released Wikidata query logs was presented. It highlighted the presence of their most prominent query fragment, i.e. C2RPQs. This fragment corresponds to highly complex recursive queries with joins and property paths. Apart from simple counting measures on this fragment, the focus was on tailoring a property path analysis and shape analysis to these queries .

Even though Bielefeldt et al. [BGK18] rightfully note the difficulty of obtaining stable observations from these query logs due to massive presence of robotic traffic, several similarities can be discovered, like low hypertreewidth and structure of property paths seem to be relatively consistent between the present study and previous work.

Additionally, while investigating on a newly discovered fragment (C2RPQs), entirely new observations were made across the distinction in terms of *Valid*

and *Unique* logs, further segmented into RoboticOK and OrganicOK, the addition of timeout logs never analyzed before, which led to add interesting dimensions to the analysis.

Main Findings We succinctly report the main findings of the analysis regarding the guiding research questions. About the distributions of sizes, we see that the RoboticOK queries are less skewed in terms of sizes than the OrganicOK queries, and this also applies to OrganicTO and RoboticTO queries, that are also inherently more complex in terms of sizes than the RoboticOK plus OrganicOK queries.

Next, property paths occur in these logs 57x more than in previously analyzed logs, which made us focus on the most representative query fragment C2RPQs. Notice that a bigger variety of property path classes again occurs with Organic queries rather than with Robotic queries, and that they exhibit a rather different structure. In fact, while the most occurring transitive classes of property paths for Robotic queries are a^*, a^+ and ab^* (thus with single alphabet symbols), the most occurring transitive classes of property paths for Robotic queries contains unions of (at least two) symbols, thus corresponding to AB^* and A^*. Such a discovery might spur interesting query processing and query optimization questions around C2RPQs, which were not addressed for the much simpler fragments of CQ. For instance, landmarking indexes have been introduced for one of the prominent classes of Robotic queries (A^*) in Valstar et al. [VFY17], but also the other prominent classes need attention when designing indexes for C2RPQs.

In the analysis, the question on the prominence of CQs and C2RPQs in these logs compared to other logs was addressed, thus bringing to the surface the most occurring recursive fragment of C2RPQs enriched with The shape classification was run with and without constants for several fragments ranging between the class of CQs and C2RPQ+ by considering/excluding constants. We essentially see that stars are a common shape; trees/forests are very common. And even 25% organic and 36% robotic queries disintegrate to a single node if constants are removed. The shape analysis with or without constants also led us to identify shifts in the shape classes due to removal of constants that are worth looking at. Constants have been disregarded in the study of C2RPQs, where indexing techniques have mainly considered the labeled paths as key index terms. The combination of indexing techniques looking at constants and labeled paths could thus be a direction to pursue in future studies on indexing structures and index maintenance for these queries [Bon+18]. The timeout queries are also interesting because they are on average greater in size and more cyclic.

Concerning tree- and hypertreewidth, the logs strongly confirm a hypothesis that is often stated in theoretical research: the cyclic queries in practical applications are only *mildly cyclic*. This means that database queries typically do not have large k-cliques encoded in their shape, but remain tree-like.

Benefits of AST Similarity Classification Next, a novel Wikidata-specific query similarity search allows to efficiently navigate the query logs starting from an initial query that users have at their disposal. This search improves the usability of the Wikidata query logs for both recursive and non-recursive queries, and it is valuable for identifying subsets of similar queries on which further assessment is possible.

After pre-processing, i.e., computing the *Valid* and *Unique* data sets, the entire analysis of the Wikidata query logs (not including the query similarity search, which has been separately measured) takes roughly 12 hours on a 24 core machine with a 2.6 GHz CPU and 128 GB RAM.

Future Isolating complex query fragments and studying suitable sophisticated metrics can be valuable for the community and may lead to further studies and assessment of these logs. This work can serve as a basis for researchers to find further interesting fragments of queries to study; it is only reported what was found in the analysis performed in the logs. However, one should always keep in mind that we are looking at *specific query logs*. It cannot be concluded from this study that a given fragment, operator, or type of query is *not* interesting to study.

7.3 Final Words

Large query logs amounting to around half a billion queries have been thoroughly studied under a multitude of aspects. Both a diversified set from multiple different sources spanning several years and more homogenous logs from Wikidata with queries from users or bot-generated traffic were taken into consideration.

The analysis of the logs both took existing work as starting point for investigating and introduced new novel techniques to classify queries and interpret the composition of queries.

The study started with simple existing measures such as operator composition, measuring of triple sizes, examining features such as subquery and projection. It analyzed properties and property paths and used it as basis for categorizing queries into fragments that take the potential complexity of

queries into consideration. Based on this, the shape of queries and their complexity as well as their hypertree representation for more complex queries is taken into consideration. A notable, although not surprising, result of all this is that a majority of queries are very simple, but there is a more interesting fraction of queries that can still be covered quite reliably with known properties. A part of this fraction and an even smaller set of remaining queries can be interesting to study under different aspects in the future.

Several approaches to problems yielded completely novel, unique results, such as the shape analysis with the identification of flower-shaped queries, the temporal analysis of streaks in logs, and the parse-tree-based similarity measure for query search. These results are especially interesting since they offer the opportunity for more interesting research in the future, since they can be used as base for new techniques.

As it has been mentioned before, the results are entirely reproducible, since the software that was written to obtain the results is made available under an open-source license, and the same input logs such as the ones from Wikidata are readily available or can be obtained from their respective sources for checking. The software framework that was developed can be used as basis for future work. It already has uses that go beyond just reproducing results, i.e. it can be used to search for similar queries, produce statistics, inspect the most popular queries, or it can just be used for exploration of logs visually and based on query characteristics.

Bibliography

[AB17] N. A. Arafat and S. Bressan. “Hypergraph Drawing by Force-Directed Placement”. In: *DEXA*. 2017.

[Abe+16a] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. “EmptyHeaded: A Relational Engine for Graph Processing”. In: *International Conference on Management of Data (SIGMOD)*. 2016, pp. 431–446.

[Abe+16b] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. “Old techniques for new join algorithms: A case study in RDF processing”. In: *International Conference on Data Engineering (ICDE) Workshops*. 2016, pp. 97–102.

[ACP12] M. Arenas, S. Conca, and J. Pérez. “Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard”. In: *World Wide Web Conference (WWW)*. 2012, pp. 629–638.

[ADK16] M. Arenas, G. I. Diaz, and E. V. Kostylev. “Reverse Engineering SPARQL Queries”. In: *Proceedings of the 25th International Conference on World Wide Web*. WWW ’16. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, pp. 239–249. ISBN: 978-1-4503-4143-1. DOI: 10.1145/2872427.2882989. URL: https://doi.org/10.1145/2872427.2882989.

[Alj+14] S. Aljaloud, M. Luczak-Roesch, T. Chown, and N. Gibbins. *Get All, Filter Details – On the Use of Regular Expressions in SPARQL Queries*. 4th USEWOD Workshop on Usage Analysis and the Web of Data. 2014.

[Ari+11] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. “An Empirical Study of Real-World SPARQL Queries”. In: *CoRR* abs/1103.5043 (2011).

[Bag+17] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. "gMark: Schema-Driven Generation of Graphs and Queries". In: *IEEE Trans. Knowl. Data Eng.* 29.4 (2017), pp. 856–869.

[Bar+18] P. Barceló, M. Kröll, R. Pichler, and S. Skritek. "Efficient Evaluation and Static Analysis for Well-Designed Pattern Trees with Projection". In: *ACM Trans. Database Syst.* 43.2 (2018), 8:1–8:44.

[Bar16] P. Barceló. `https://databasetheory.org/node/47`. 2016.

[BBE17] M. Bannach, S. Berndt, and T. Ehlers. "Jdrasil: A Modular Library for Computing Tree Decompositions". In: *16th International Symposium on Experimental Algorithms (SEA)*. 2017, 28:1–28:21.

[BBG13] G. Bagan, A. Bonifati, and B. Groz. "A trichotomy for regular simple path queries on graphs". In: *Principles of Database Systems (PODS)*. 2013, pp. 261–272.

[BDG07] G. Bagan, A. Durand, and E. Grandjean. "On Acyclic Conjunctive Queries and Constant Delay Enumeration". In: *Computer Science Logic (CSL)*. 2007, pp. 208–222.

[BGK18] A. Bielefeldt, J. Gonsior, and M. Krötzsch. "Practical Linked Data Access via SPARQL: The Case of Wikidata". In: *Workshop on Linked Data (LDOW)*. 2018.

[Bie+18] A. Bielefeldt, J. Gonsior, L. Gonzalez, M. Krötzsch, and S. Malyshev. *Wikidata SPARQL Logs*. `https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en`. Sept. 2018.

[BLS19] D. Baelde, A. Lick, and S. Schmitz. "Decidable XPath Fragments in the Real World". In: *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 2019, pp. 285–302. DOI: `10.1145/3294052.3319685`. URL: `https://doi.org/10.1145/3294052.3319685`.

[BLS99] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: A Survey*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-432-X.

[BMT17a] A. Bonifati, W. Martens, and T. Timm. "An Analytical Study of Large SPARQL Query Logs". In: *PVLDB* 11.2 (2017), pp. 149–161. DOI: `10.14778/3167892.3167895`.

[BMT17b] A. Bonifati, W. Martens, and T. Timm. "An Analytical Study of Large SPARQL Query Logs". In: *CoRR* abs/1708.00363 (2017).

[BMT18] A. Bonifati, W. Martens, and T. Timm. "DARQL: Deep Analysis of SPARQL Queries". In: *The Web Conference (WWW)*. 2018, pp. 187–190. DOI: 10.1145/3184558.3186975.

[BMT19a] A. Bonifati, W. Martens, and T. Timm. "An Analytical Study of Large SPARQL Query Logs". In: *The VLDB Journal* (2019). Extended version of [BMT17a]. DOI: 10.1007/s00778-019-00558-9.

[BMT19b] A. Bonifati, W. Martens, and T. Timm. "Navigating the Maze of Wikidata Query Logs". In: *The Web Conference (WWW)* (2019). DOI: 10.1145/3308558.3313472.

[BMT19c] A. Bonifati, W. Martens, and T. Timm. "SHARQL: Shape Analysis of Recursive SPARQL Queries". Unpublished. 2019.

[Bon+18] A. Bonifati, G. H. L. Fletcher, H. Voigt, and N. Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[BPS15] P. Barceló, R. Pichler, and S. Skritek. "Efficient Evaluation and Approximation of Well-designed Pattern Trees". In: *Principles of Database Systems (PODS)*. 2015, pp. 131–144. DOI: 10.1145/2745754.2745767. URL: http://doi.acm.org/10.1145/2745754.2745767.

[CGH95] D. M. Chickering, D. Geiger, and D. Heckerman. "On Finding a Cycle Basis with a Shortest Maximal Cycle". In: *Inf. Process. Lett.* 54.1 (Apr. 1995), pp. 55–58. ISSN: 0020-0190. DOI: 10.1016/0020-0190(94)00231-M. URL: http://dx.doi.org/10.1016/0020-0190(94)00231-M.

[CH19] S. Cheng and O. Hartig. "OPT+: A Monotonic Alternative to OPTIONAL in SPARQL". In: *Journal of Web Engineering* 48 (2019), pp. 169–206. URL: https://doi.org/10.13052/jwe1540-9589.18135.

[CM77] A. Chandra and P. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In: *Symposium on the Theory of Computing (STOC)*. 1977, pp. 77–90.

[CR97] C. Chekuri and A. Rajaraman. "Conjunctive Query Containment Revisited". In: *International Conference on Database Theory (ICDT)*. 1997, pp. 56–70.

[Cze+15] W. Czerwiński, W. Martens, P. Parys, and M. Przybylko. "The (Almost) Complete Guide to Tree Pattern Containment". In: *ACM Symposium on Principles of Database Systems (PODS)*. 2015, pp. 117–130.

[Cze+18] W. Czerwiński, W. Martens, M. Niewerth, and P. Parys. "Minimization of Tree Patterns". In: *J. ACM* 65.4 (2018), 26:1–26:46.

[DBP17] DBPedia. `http://wiki.dbpedia.org/datasets`. 2017.

[Fär+15] M. Färber, B. Ell, C. Menne, and A. Rettinger. "A Comparative Survey of DBpedia, Freebase, OpenCyc, Wikidata, and YAGO". In: *Semantic Web Journal, July* (2015).

[FGP18] W. Fischl, G. Gottlob, and R. Pichler. "General and Fractional Hypertree Decompositions: Hard and Easy Cases". In: *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18. Houston, TX, USA: ACM, 2018, pp. 17–32. ISBN: 978-1-4503-4706-8. DOI: 10.1145/3196959.3196962. URL: http://doi.acm.org/10.1145/3196959.3196962.

[Fis+19] W. Fischl, G. Gottlob, D. M. Longo, and R. Pichler. "HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings". In: *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '19. Amsterdam, Netherlands: ACM, 2019, pp. 464–480. ISBN: 978-1-4503-6227-6. DOI: 10.1145/3294052.3319683. URL: http://doi.acm.org/10.1145/3294052.3319683.

[Fis18] W. Fischl. *newdetkdecomp*. `https://github.com/TUfischl/newdetkdecomp`. Visited on September 10th, 2018. 2018.

[FVY17] G. H. L. Fletcher, H. Voigt, and N. Yakovets. "Declarative Graph Querying in Practice and Theory". In: *International Conference on Extending Database Technology (EDBT)*. 2017, pp. 598–601.

[GGS14] G. Gottlob, G. Greco, and F. Scarcello. "Treewidth and Hypertree Width". In: *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2014, pp. 3–38.

[GLS01] G. Gottlob, N. Leone, and F. Scarcello. "The complexity of acyclic conjunctive queries". In: *J. ACM* 48.3 (2001), pp. 431–498.

[GLS02] G. Gottlob, N. Leone, and F. Scarcello. "Hypertree Decompositions and Tractable Queries". In: *J. Comput. Syst. Sci.* 64.3 (2002), pp. 579–627.

[GMS09] G. Gottlob, Z. Miklós, and T. Schwentick. "Generalized Hypertree Decompositions: NP-hardness and Tractable Variants". In: *J. ACM* 56.6 (Sept. 2009), 30:1–30:32. ISSN: 0004-5411. DOI: 10.1145/1568318.1568320. URL: http://doi.acm.org/10.1145/1568318.1568320.

[Got+16] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. "Hypertree Decompositions: Questions and Answers". In: *Principles of Database Systems (PODS)*. 2016, pp. 57–74.

[GS09] G. Gottlob and M. Samer. "A Backtracking-based Algorithm for Hypertree Decomposition". In: *J. Exp. Algorithmics* 13 (Feb. 2009), 1:1.1–1:1.19. ISSN: 1084-6654. DOI: 10.1145/1412228.1412229. URL: http://doi.acm.org/10.1145/1412228.1412229.

[Han+16] X. Han, Z. Feng, X. Zhang, X. Wang, G. Rao, and S. Jiang. "On the statistical analysis of practical SPARQL queries". In: *WebDB*. 2016, p. 2.

[Her+16] D. Hernández, A. Hogan, C. Riveros, C. Rojas, and E. Zerega. "Querying Wikidata: Comparing SPARQL, Relational and Graph Databases". In: *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*. 2016, pp. 88–103. DOI: 10.1007/978-3-319-46547-0_10.

[HP15] J. Huelss and H. Paulheim. "What SPARQL Query Logs Tell and Do Not Tell About Semantic Relatedness in LOD — Or: The Unsuccessful Attempt to Improve the Browsing Experience of DBpedia by Exploiting Query Logs". In: *ESWC Satellite Events*. 2015, pp. 297–308.

[HS13] S. Harris and A. Seaborne. *SPARQL 1.1 query language*. Tech. rep. https://www.w3.org/TR/2013/REC-sparql11-query-20130321. World Wide Web Consortium (W3C), Mar. 2013.

[IUV17] M. Idris, M. Ugarte, and S. Vansummeren. "The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates". In: *International Conference on Management of Data (SIGMOD)*. 2017, pp. 1259–1274.

[Jag+07] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. "Making database systems usable". In: *International Conference on Management of Data (SIGMOD)*. 2007, pp. 13–24.

[KEK17] O. Kalinsky, Y. Etsion, and B. Kimelfeld. "Flexible Caching in Trie Joins". In: *International Conference on Extending Database Technology (EDBT)*. 2017, pp. 282–293.

[KK16] M. Kaminski and E. V. Kostylev. "Beyond Well-designed SPARQL". In: *International Conference on Database Theory (ICDT)*. 2016, 5:1–5:18.

[KK18] M. Kaminski and E. V. Kostylev. "Complexity and Expressive Power of Weakly Well-Designed SPARQL". In: *Theory Comput. Syst.* 62.4 (2018), pp. 772–809.

[KPS16] M. Kröll, R. Pichler, and S. Skritek. "On the Complexity of Enumerating the Answers to Well-designed Pattern Trees". In: *International Conference on Database Theory (ICDT)*. 2016, 22:1–22:18.

[KS08] B. Kimelfeld and Y. Sagiv. "Revisiting redundancy and minimization in an XPath fragment". In: *International Conference on Extending Database Technology (EDBT)*. 2008, pp. 61–72.

[KV98] P. G. Kolaitis and M. Y. Vardi. "Conjunctive-Query Containment and Constraint Satisfaction". In: *Symposium on Principles of Database Systems (PODS)*. 1998, pp. 205–213.

[LDB] LDBC. `http://ldbcouncil.org`.

[Let+13] A. Letelier, J. Pérez, R. Pichler, and S. Skritek. "Static analysis and optimization of semantic web queries". In: *ACM Trans. Database Syst.* 38.4 (2013), 25:1–25:45.

[Lev66] V. I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet Physics Doklady* (1966), 10(8):707–710.

[LM13] K. Losemann and W. Martens. "The complexity of regular expressions and property paths in SPARQL". In: *ACM Trans. Database Syst.* 38.4 (2013), 24:1–24:39.

[LMV16a] L. Libkin, W. Martens, and D. Vrgoc. "Querying Graphs with Data". In: *J. ACM* 63.2 (2016), 14:1–14:53.

[LMV16b] L. Libkin, W. Martens, and D. Vrgoč. "Querying Graphs with Data". In: *J. ACM* 63.2 (Mar. 2016), 14:1–14:53. ISSN: 0004-5411. DOI: 10.1145/2850413. URL: http://doi.acm.org/10.1145/2850413.

[LSQ18] LSQ. http://aksw.github.io/LSQ/. 2018.

[Mal+18] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. "Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph". In: *International Semantic Web Conference (ISWC)*. 2018, pp. 376–394.

[MN07] W. Martens and J. Niehren. "On the minimization of XML Schemas and tree automata for unranked trees". In: *J. Comput. Syst. Sci.* 73.4 (2007), pp. 550–583.

[Möl+10] K. Möller, M. Hausenblas, R. Cyganiak, S. Handschuh, and G. Grimnes. "Learning from linked open data usage: Patterns & metrics". In: *Web Science Conference (WSC)*. 2010.

[Mor+11] M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. "DBpedia SPARQL Benchmark — Performance Assessment with Real Queries on Real Data". In: *International Semantic Web Conference (ISWC)*. 2011, pp. 454–469.

[MS04] G. Miklau and D. Suciu. "Containment and equivalence for a fragment of XPath". In: *J. ACM* 51.1 (2004), pp. 2–45.

[MSJ19] S. Maniu, P. Senellart, and S. Jog. "An Experimental Study of the Treewidth of Real-World Graph Data". In: *22nd International Conference on Database Theory (ICDT 2019)*. Ed. by P. Barcelo and M. Calautti. Vol. 127. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 12:1–12:18. ISBN: 978-3-95977-101-6. DOI: 10.4230/LIPIcs.ICDT.2019.12. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10314.

[MT17] W. Martens and T. Trautner. "Enumeration Problems for Regular Path Queries". In: *CoRR* abs/1710.02317 (2017).

[MT18a] W. Martens and T. Trautner. "Evaluation and Enumeration Problems for Regular Path Queries". In: *International Conference on Database Theory (ICDT)*. 2018, 19:1–19:21.

[MT18b] W. Martens and T. Trautner. "Evaluation and Enumeration Problems for Regular Path Queries". In: *21st International Conference on Database Theory (ICDT 2018)*. Ed. by B. Kimelfeld and Y. Amsterdamer. Vol. 98. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 19:1–19:21. ISBN: 978-3-95977-063-7. DOI: 10.4230/LIPIcs.ICDT.2018.19. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8594.

[Neo17] Neo4j. http://www.opencypher.org/ocig2. Visited on July 6th, 2017. 2017.

[NJ11] A. Nandi and H. V. Jagadish. "Guided Interaction: Rethinking the Query-Result Paradigm". In: *PVLDB* 4.12 (2011), pp. 1466–1469.

[NW10] T. Neumann and G. Weikum. "The RDF-3X engine for scalable management of RDF data". In: *VLDB J.* 19.1 (2010), pp. 91–113.

[PA15a] M. Pawlik and N. Augsten. "Efficient Computation of the Tree Edit Distance". In: *ACM Trans. Database Syst.* 40.1 (Mar. 2015), 3:1–3:40. ISSN: 0362-5915. DOI: 10.1145/2699485. URL: http://doi.acm.org/10.1145/2699485.

[PA15b] M. Pawlik and N. Augsten. "Tree edit distance: Robust and memory-efficient". In: *Information Systems* 56 (Aug. 2015). DOI: 10.1016/j.is.2015.08.004.

[PAG09] J. Pérez, M. Arenas, and C. Gutierrez. "Semantics and complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (2009), 16:1–16:45. DOI: 10.1145/1567274.1567278. URL: http://doi.acm.org/10.1145/1567274.1567278.

[PAG10] J. Pérez, M. Arenas, and C. Gutiérrez. "nSPARQL: A navigational language for RDF." In: *J. Web Semant.* 8.4 (2010), pp. 255–270. URL: http://dblp.uni-trier.de/db/journals/ws/ws8.html#PerezAG10.

[Pis18] A. Piscopo. "Wikidata: A New Paradigm of Human-Bot Collaboration?" In: *arXiv preprint arXiv:1810.00931* (2018).

[Pos18] PostgreSQL Global Development Group. *PostgreSQL*. http://www.postgresql.org. 2018.

[PV11] F. Picalausa and S. Vansummeren. "What are real SPARQL queries like?" In: *International Workshop on Semantic Web Information Management (SWIM)*. 2011, pp. 1–7.

[RH14] L. Rietveld and R. Hoekstra. "Man vs. Machine: Differences in SPARQL Queries". In: *ESWC 2014*. 2014.

[Sal+15] M. Saleem, I. Ali, A. Hogan, Q. Mehmood, and A.-C. Ngonga Ngomo. "LSQ: The Linked SPARQL Queries Dataset". In: *International Semantic Web Conference (ISWC)*. 2015, pp. 261–269.

[Sam18] M. Samer. ***detkdecomp***. `https://github.com/daajoe/detkdecomp`. Visited on September 10th, 2018. 2018.

[SO87] D. S. Johnson and H. O. Pollak. "Hypergraph planarity and the complexity of drawing Venn diagrams". In: *Journal of Graph Theory* 11 (Sept. 1987), pp. 309–325. DOI: `10.1002/jgt.3190110306`.

[Sys17] Systap. *Blazegraph*. `http://www.blazegraph.com`. Visited on June 12th, 2017. 2017.

[Tim18] T. Timm. ***DARQL***. `https://github.com/PoDMR/darql/`. 2018.

[Tim19a] T. Timm. ***SHARQL***. `https://github.com/PoDMR/sharql/`. 2019.

[Tim19b] T. Timm. `https://podmr.github.io/darql/property-sunburst/`. Created February 2019. 2019.

[VFY17] L. D. J. Valstar, G. H. L. Fletcher, and Y. Yoshida. "Landmark Indexing for Evaluation of Label-Constrained Reachability Queries". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 2017, pp. 345–358.

[Vid+10] M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. "Efficiently Joining Group Patterns in SPARQL Queries". In: *Extended Semantic Web Conference (ESWC)*. 2010, pp. 228–242.

[VK14] D. Vrandecic and M. Krötzsch. "Wikidata: a free collaborative knowledgebase". In: *Commun. ACM* 57.10 (2014), pp. 78–85.

[Wik17] Wikidata. `http://wikidata.org`. Visited on February 8th, 2017. 2017.

[Wik18] Wikimedia Foundation. `https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`. Visited on May 4th, 2017. 2018.

[Yan81] M. Yannakakis. “Algorithms for Acyclic Database Schemes”. In: *International Conference on Very Large Data Bases*. 1981, pp. 82–94.

List of Figures

List of Tables

List of Notations

$!a$: Negated property set of a, 39

T_p: Subtree of T rooted at p, 71

χ: Vertex mapping in decompositions, 70

λ: Hyperedge mapping in decompositions, 71

$vars(P)$: variables occuring in pattern P, 17

$\hat{}a$: Inverse path of a, 39

C2RPQ+ : Extended Conjunctive 2-way Regular Path Queries, 55

$\mathsf{C2RPQ_F}$: Conjunctive 2-way Regular Path Queries with Filter, 55

$\mathsf{C2RPQ_{OF}}$: Conjunctive 2-way Regular Path Queries with Filter and Optional, 55

C2RPQ : Conjunctive 2-way Regular Path Queries, 18

$\mathsf{CQ_F}$: Conjunctive Queries with Filter, 25

$\mathsf{CQ_{OFV}}$: Conjunctive Queries extended with Filter, Optional, and Values, 54

$\mathsf{CQ_{OF}}$: Conjunctive Queries with Filter and Optional, 53

$\mathsf{CQ_{OF+}}$: Extended Conjunctive Query fragment, 54

CQ : Conjunctive Queries, 18

OK : Combined organic and robotic queries without timeout, 14

OrganicOK : Organic queries without timeout, 14

OrganicTO : Organic queries with timeout, 14

RoboticOK : Robotic queries without timeout, 14

RoboticTO : Robotic queries with timeout, 14

TO : Combined organic and robotic queries with timeout, 14

Index

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass ich die Hilfe von gewerblichen Promotionsberatern bzw. –vermittlern oder ähnlichen Dienstleistern weder bisher in Anspruch genommen habe, noch künftig in Anspruch nehmen werde.

Zusätzlich erkläre ich hiermit, dass ich keinerlei frühere Promotionsversuche unternommen habe.