A HYBRID FRAMEWORK FOR QUERYING LINKED DATA DYNAMICALLY

JÜRGEN UMBRICH

Dissertation
submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

September 2012

Supervisors: **Prof. Dr. Stefan Decker** and **Priv.-Doz. Dr. Axel Polleres**

Internal Examiner: **Prof. Dr. Manfred Hauswirth**

External Examiner: **Prof. Dr. Claudio Gutiérrez**

Digital Enterprise Research Institute
National University of Ireland, Galway / Ollscoil na hÉireann, Gaillimh

The research presented herein was supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

ABSTRACT

As of today, the Web has evolved to become the largest collection of information made available by mankind. Researchers and developers are continuously working on transforming this loosely connected data collection into a giant knowledge base. As part of this trend, the Semantic Web community has started a movement to transform the Web of unstructured text into the so called "Web of Data" – a framework to create, share and reuse data by humans and machines alike across application, enterprise, and community boundaries. From this movement, Linked Data has emerged as a set of best practices to publish, connect and discover structured data on the Web using standard formats. As of today, there are over thirty billion public facts which can be accessed, reused and combined by individuals as well as organisations and companies.

As the Web of Data continues to expand and diversify, it becomes more and more dynamic with data being constantly generated, removed and updated, e.g., from sensor/stream sources. New querying techniques are required to efficiently keep up with this trend. While traditional approaches facilitate fast query times by replicating Web data in optimised offline index structures , they cannot deal efficiently with dynamic data and cannot guarantee up-to-date results. A new generation of distributed Linked Data query engines address this problem and deliver up-to-date results by retrieving query relevant data immediately before or during query execution. However fetching data at runtime from potentially hundreds or thousands of relevant Web sources is slow compared to optimised index lookups.

This thesis studies and improves distributed query approaches for Linked Data and develops a hybrid query framework that offers fresh and fast query results by combining centralised and distributed query techniques with a novel query planning approach based on knowledge about the dynamicity of data.

We start by identifying the different levels of dynamicity within Linked Data and highlight the challenges for centralised query approaches to deliver up-to-date results if operating over such dynamic data. We then present a study of link traversal based query execution approaches for Linked Data and show how the query performance can be improved by providing reasoning extensions. We have also developed an approximate index structure that summarises the graph-structured content of Web sources, and provide an algorithm that exploits this source summary index. Finally, we propose and evaluate a novel hybrid query engine framework that combines the execution strength of materialised query approaches with the live results from distributed query approaches. The query planning phase uses a cost-model that combines standard selectivity and novel dynamicity estimates to enable fast and fresh results.

DECLARATION

I declare that this thesis is composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Galway, Ireland, September 2012

Jürgen Umbrich

ACKNOWLEDGMENTS

One of the joys at the end of writing a dissertation is to reflect the journey past and use the opportunity to acknowledge all the people who have helped and supported me along this long but fulfilling venture.

First and foremost, I would like to express my sincere gratitude to my supervisors Axel and Stefan for their support, enthusiastic encouragement and useful critiques of this research work. I would also like to thank my examiners Manfred Hauswirth and Claudio Gutiérrez for their critical review of this dissertation and their valuable feedback and discussion during the viva.

During the last couple of years, I had the great pleasure to meet, collaborate and be accompanied by many great minds and people. A very special thank to my mentors Marcel and Michael for all their time, inspiring discussions and the many fruitful collaborations. I own my deepest gratitude to my colleague Aidan. I was fortunate enough to work with him in several projects, he was always a great source of inspiration during that time. I would also like to take this opportunity to thank all my colleagues in DERI and all my co-author for – with very sincere gratitudes to Andreas, Katja, Kai and Tobias. In addition, it was great moral motivation to share the write up phase with my colleagues Nuno and Laura.

A very special thank to Josi for all her help and support in a number of ways during the last years. And last but not least, I would like to thank my family for all their support, understanding and patience.

CONTENTS

1 INTRODUCTION

For the last decades, the Internet has been a widely used medium to share information and knowledge in our everyday life. At present, the role of internet users transformed from solely consumers or producers into "prosumer" as already envisioned by Toffler [1980]; human and software agents, not only consume information but also create, publish, share and distribute information in real-time. As a consequence the Web has evolved to become the largest existing collection of information made available by mankind.

Researchers and developers are continuously working on transforming this connected data collection into a giant knowledge base bearing the potential for (as of yet) unthought-of applications. A crucial part of this picture are efficient (access) mechanisms to discover, search and query relevant information. Search and query tasks become even more important and challenging if we search over dynamically changing data, whereas up-to-date/fresh information is a necessary requirement.

1.1 MOTIVATION

Search and query components for Web data are of high importance and their relevance is undeniable; this can be justified, for instance, by the huge success of search engines such as Google, Yahoo and Microsoft. In the early stage of the Web (around 1992), the index of available Web pages was manually maintained as a list of Web servers, such as the W3Catalog, developed by Nierstrasz [1996]. Later, due to the increase number of Web servers, this server list was managed semi-automatically with simple scripts. The demand for accurate and efficient search components increased with the steady growth of the Web. In 1997, the now ubiquitous Google engine went online and revolutionised search on the Web with a new result ranking algorithm called PageRank [Brin and Page, 1998; Page et al., 1999]. Interestingly, this algorithm exploits the fact that Web documents are connected/related by hyperlinks and takes advantage of the underlying graph structure of the data.

Information on the Web was mainly produced by humans for the consumption of humans and not for machines [Henzinger et al., 2002; Lee et al., 2001]. Human readable information is mostly presented as free text embedded in documents. Providing efficient and accurate search and query capabilities over such a collection of free text poses major challenges. Web search still demands time consuming human interaction (e.g. to refine search requests [Press, 1999; Wildemuth, 2004], disambiguate results, or to perform multilingual search [Buitelaar and Cimiano, 2008]). Surprisingly, more and more Web content nowadays stems from structured information in databases but still is externalised on the Web in a semi-structured and mostly human readable way [Senellart et al., 2008].

At the end of the 20th century, the Semantic Web research community began focusing on overcoming these problem by transforming the Web of unstructured text into the so called "Web of Data" – a framework to create, share and reuse data by humans and machine alike across application, enterprise, and community boundaries [Berners-Lee, 1998]. Semantic Web technology is built on top of the Resource Description Framework (RDF), a standard data format which allows to describe resources (anything with identity) and their relations in a structured and machine understandable way. Therefor automated data-processing algorithms can

be developed and applied. We can witness the benefit of structured data in many tasks, especially in the context of search and querying. Major search engines are actively exploiting and promoting the use of structured data to improve search functionalities and the quality of search results [Baeza-Yates et al., 2008; Bizer et al., 2009; Chang et al., 2008]. For example, a modern Web search engine is now able to understand that a particular text snippet refers to the temperature information for a city, in a forecast page, or can identify the latest price of a car for sale in an advertising page. As well as returning a list of documents for a user search, results now also contain direct answers to a query such as the weather forecast for a city, the rating of a movie and the play times in the local theatres, or the contact information and location for restaurants or shops.

Recently, the Linked Data community has emerged around a set of best practices to publish, connect and discover structured data on the Web. One core principle of Linked Data is to use HTTP URIs to refer to real world things and a standardised data format (RDF) to describe those things so that machines can retrieve data about named things from the Web. Another core principle is to describe relations between things by links; thus, published Linked Data forms a "Giant Global Graph" of data. The result is a concrete instantiation of a Web of Data as envision by Berners-Lee [1998], which can be also seen as a global decentralised database [Hausenblas and Karnstedt, 2010]. Proponents of Linked Data encourage people, communities, organisations [O'Riain et al., 2012] and governments [Maali et al., 2012] to publish data by a set of standard guidelines. As of today, the Web contains estimated 30 billion facts about real world things, and their relations, published as RDF on the Web [Bizer et al., 2011].

Another recent trend on the Web, and one which poses a novel challenge for search engines, is that more people and especially devices and sensors have access to and become part of the Web. The Web itself can be seen as a self-organising ecosystem; many units participate in a parallel and distributed manner by creating, publishing but also interlinking information, and subsequently creating a highly dynamic environment in which content and information can be created, removed and changed at anytime. In such a system, more data gets created and becomes more dynamic with an increasing number of participants. In particular, participants such as mobile devices or sensors create and publish data which is only valid for a particular time period (e.g., current location, temperature or seismic activities).

We can expect even more dynamic content and data growth if we consider the current trend in the mobile application market and the effort of integrating sensor data into the Web of Data. A report from Gartner in 2005 predicted the generation of terabytes of data per day with the expected integration of sensor data from articles of daily use (e.g., from cars, kitchen utilities or from personal or health devices) [Raskino et al., 2005]. Notably, the integration of sensor data for environmental monitoring will increase the demand for efficient real-time search and query solutions over structured data.

An increasingly important challenge is to reflect the outcome of these "change events" in search results, particularly delivered to the end users or applications for which the time validity of information is a crucial and compulsory requirement. Though various Web search engines have shown the power and potential of centralisation, even the preeminent Google machinery struggles to give up-to-date answers over vast amounts of dynamic sources. Linked Data presents new opportunities in this regard: a new set of query engines exploit that URI names appearing in user queries also correspond to addresses from which up-to-date data can be found.

1.2 PROBLEM STATEMENT

As the Web of Data continues to expand and diversify, and as it becomes more dynamic, new querying techniques are required to keep up with its developments. While existing approaches offer either fast query times by using optimised index structures or up-to-date result by processing a query directly over the Web, there exists no satisfying solution to efficiently query dynamic Linked Data and guarantee up-to-date results at the same time.

Standard approaches for querying Linked Data rely on data warehousing or materialisation-based approaches similar to traditional Web search engines [Bishop et al., 2011b; Erling and Mikhailov, 2009; Oren et al., 2008]. These approaches locally store Web data in optimised index structures by retrieving and parsing the content of Web sources. The benefits of these pre-processing and replication steps are excellent query response times and scalability. However, as we will see in this thesis, these approaches cannot efficiently deal with dynamic data and usually cannot guarantee that the returned answers reflect the current state of the information on the Web; maintaining comprehensive and up-to-date local indexes is a nearly impossible task due to the size of the Web and the change rate of information [Cho and Garcia-Molina, 2003b; Ntoulas et al., 2004].

A new generation of Linked Data query engines addresses the problem of outdated query results by exploiting the fact that Linked Data itself can be conceptualised as a "*heterogeneous distributed database*" spanning the Web [Hausenblas and Karnstedt, 2010]. These decentralised query approaches directly process queries over this global database by traversing the global data graph and return nearly real-time query answers by retrieving data directly from remote Web sources immediately before or during query processing. However fetching data at runtime from potentially hundreds or thousands of relevant Web sources is an expensive and time consuming operation compared to optimised index lookups and also requires careful attention to not overload Web servers with HTTP requests. Moreover, and depending on the actual approach, the query engine can only execute certain types of queries and has possibly limited knowledge about the sources on the Web which might affect the result recall.

There exist different variations to select the query relevant sources for these decentralised query techniques. Hartig et al. [2009] proposed a link traversal based query execution approach to discover the relevant sources in an explorative way by following data links during query execution. This approach requires no knowledge about sources but cannot execute arbitrary queries and potentially misses answers if relevant content is available in sources which cannot be discovered because of missing links.

Other approaches store the (possibly summarised) content of millions of Web sources in specific index structures prior to query execution and exploit this knowledge to determine query relevant sources either directly before or during the query execution [Ladwig and Tran, 2010; Li and Heflin, 2010]. While these indexes might increase the number of query relevant sources, they require similar preprocessing steps and resource requirements as materialisation based approaches.

There are also query approaches that combine materialisation-based and traversal based Linked Data querying [Hartig, 2011a; Ladwig and Tran, 2011]. These solutions primarily use a local index as a cache to serve answers and only retrieves remote data from sources not known to the cache. In general, this mixed approach facilitates better query times than the approaches which only rely on data retrieved from the Web at query time. However, the drawback is again the problem of maintaining a local index which potentially leads to inconsistent results compared to the Web.

Without question, there exists an inherent trade-off between query approaches that give fresh results versus approaches that give fast results. Even the existing mixed approaches still rely heavily on materialised indexes and do not explicitly explored this core trade-off of *fresh vs. fast results*.

1.3 RESEARCH HYPOTHESES

The core goal of this thesis is to demonstrate that it is possible to efficiently process SPARQL queries over Linked Data and deliver up-to-date results even considering that Linked Data is often highly dynamic.

The assumption that Linked Data is (in significant parts) dynamic is very likely to hold, considering that studies about the traditional Web reported about strong underlying dynamics [Brewington and Cybenko, 2000b; Cho and Garcia-Molina, 2003a; Lim et al., 2001]. However, there exists only limited empirical evidence to verify these strong assumptions for Linked Data. Furthermore, there is no reported evidence that data warehousing approaches for Linked Data return potentially inconsistent results with regards to the actual data available on the Web at query time.

Moreover, and towards efficient Linked data querying, the potential of state-of-the-art link traversal based query solutions is not fully exploited and can be further extended by applying reasoning over the query relevant data, similar to traditional Linked Data data warehousing approaches [Delbru et al., 2011; Hogan, 2011]. In addition, we can optimise Linked Data querying using specialised index structures which summarise the content of Web sources in a very compact way.

None of the state-of-the-art approaches exploit the fact that parts of the data on the Web are never or rarely changing, whereas other parts are undergoing frequent changes. Thus, a hybrid query engine can be developed to fully exploit the strengths of materialised and live Linked Data query solutions – i.e., facilitating fresh *and* fast results – by understanding the dynamicity of the different facets of Linked Data and of the query relevant information being requested.

In more detail, we investigate and verify the following hypotheses:

[H1] The content of some Linked Data sources are static and others are dynamic and hence, Linked Data search engines and centralised stores based on traditional database query techniques (data warehousing) are necessarily partially outdated and potentially return stale results.

[H2] Pure traversal-based Linked Data query approaches are incomplete, expensively slow and limited to certain query types.

[H3] We can significantly improve the recall of traversal based Linked data query approaches by exploiting the semantics of RDFS and OWL, just as for data warehousing approaches [Delbru et al., 2011; Hogan, 2011].

[H4] Source selection techniques based on lightweight data summaries can improve the query times and result recall and, at the same time, relax the restrictions on supported query types in the pure traversal scenario.

[H5] We can efficiently query Linked Data and deliver up-to-date results with a hybrid query framework that combines centralised and distributed query approaches, using query planning techniques guided by knowledge about the dynamicity of Linked Data.

1.4 CONTRIBUTION

The results of this thesis are contributions to the study and improvement of query approaches for Linked Data, with the main contribution being the design and evaluation of an original hybrid query architecture for Linked Data which tackles the problem of efficiently delivering up-to-date results.

We investigate the following aspects:

(*contribution to* H1)

- We present the design and findings of two experiments which present evidence that fragments of Linked Data are dynamic and traditional materialised repositories cannot entirely keep their indexed data coherent with their Web counter-parts. These studies are the first of their kind for Linked Data.

(*contribution to* H2)

- We study in-depth link traversal based query approaches for Linked Data to show how these approaches pose practical restrictions on the type of executable queries and offer slow query times and a potentially low result recall.

(*contribution to* H3)

- We study optimisations practically addressing the two identified limitations of link traversal query execution which improve the overall query execution time by reducing the number of necessary source-accesses without affecting the number of results. In addition, several extensions are introduced which potentially improve the result recall by integrating lightweight semantics available in the source information. The benefit of these optimisations are empirically grounded and their applicability is verified with real-world queries.

(*contribution to* H4)

- We investigate several lightweight source selection approaches to further improve the query times, increase the result recall and loosen the query type restriction of pure link traversal based query approaches. As a result, we develop an approximate index structure that summarises the graph-structured content of sources, and an algorithm for answering conjunctive queries over Linked Data on the Web that exploits the source summary. Experimental results show that the lightweight index structure enables up-to-date query results over Linked Data, while keeping the overhead for querying low.

(*contribution to* H5)

- We present framework that incorporates knowledge about data dynamics into query planning to efficiently combine the execution strength of materialised query approaches with live results from distributed query approaches into a novel lightweight, hybrid query architecture. The query engine uses knowledge about the dynamics of Linked Data as statistical input for a novel query planning component that classifies parts of a query as either static or dynamic, where the static sub-query is executed over a centralised score, and the dynamic sub-query is executed using existing distributed query techniques. The query planning phase uses a cost-model that combines standard selectivity and novel dynamicity estimates to enable fast and fresh results.

1.5 IMPACT

Various parts of this thesis have been published as journal, conference and workshop articles.

We have published several contributions centred around the topic of dataset dynamics and how to assess the dynamicity of structured Web content [Umbrich et al., 2010a,b,c].

The study of the practicability of link traversal based query approaches and the proposed extensions to overcome the limitations of slow query times and potentially low result recall where published in [Umbrich et al., 2012a]. An extended version is submitted to the Semantic Web Journal and openly available to review.[1]

The study and comparison of query approaches with lightweight data summaries was originally published in [Harth et al., 2010] and extended in [Umbrich et al., 2011].

The fundamental ideas of a hybrid query engines were first put forward in [Umbrich et al., 2012d] and further developed in [Umbrich et al., 2012b]. The results of the first prototype were recently published in [Umbrich et al., 2012c].

During the time of this work, various contribution in other research areas were published.[2] We worked in the last 6 years on the architecture and data life cycle of a Semantic Web search engine, resulting in several publications such as [Harth et al., 2007a; Hogan et al., 2011, 2010, 2012b].

1.6 THESIS OUTLINE

The remainder of this thesis is structured as follows:

Chapter 2 introduces notations and core concepts, such as the RDF and SPARQL and provides a detailed overview about the state-of-the art in Linked Data query approaches and systems;

Chapter 3 introduces our evaluation dataset(s) and provides initial results verifying the dynamic nature of Linked Data.

Chapter 4 presents our optimisations, extensions and experiments for link traversal query approaches that require no prior knowledge of data sources;

Chapter 5 details our investigation of lightweight data summary methods to execute more complex queries over the Web of Data compared to link traversal based query execution ;

Chapter 6 describes our novel hybrid query technique using knowledge about dynamics and index freshness/coverage in the query planning;

Chapter 7 contains a detailed reflection upon the work in this thesis and its concrete objectives, well as and directions for future work.

1 http://semantic-web-journal.net/content/link-traversal-querying-diverse-web-data

2 See http://scholar.google.com/citations?user=Vy7mya4AAAAJ&hl=en for a full list of publications.

2 BACKGROUND & STATE OF THE ART ON LINKED DATA QUERYING

"You do not start by Adam & Eve"

— Axel Polleres, *2012*

In this chapter, we provide background and introduce the underlying concepts of the Semantic Web and Linked Data. Furthermore, we present the state of the art on querying Linked Data.

2.1 THE WORLD WIDE WEB (WWW)

The World Wide Web (WWW), or the Web for short, is a system which uses the Internet and allows anyone to publish, share and consume information at a global scale. The Web originally was proposed by Berners-Lee and Cailliau [1990]. Four years later, in 1994, the growth in uptake of the Web happened with the implementation of a Web browser, called Mosaic. The intuitive and user friendly graphical interface of the Mosaic browser are contributed to the Internet becoming mainstream. The guidelines for the success and the design of the World Wide Web were published by Jacobs and Walsh [2004] as the "*Architecture of the World Wide Web*" in 2004. The core elements of the Web are (i) identifiers in form of Uniform Resource Identifier (URI)s, (ii) interaction architectural principles such as standard protocols like the Hypertext Transfer Protocol (HTTP) and communication patterns (e.g., accessing mechanisms or availability & reliability) and (iii) unrestricted data formats.

URI A URI is a specific string to uniquely identify a resource at a global scale. The general syntax for a URI is:

`scheme://domain:port/path?query_string#fragment_id`

The `scheme` defines the access mechanism for a URI and the `port` defines the host port number for the protocol. The `domain` contains the fully qualified domain name (FQDN) of the Web server or its IP address, sometimes referred to as the *authority*. The `path` specifies the local location of the resource on the server. The remaining parts, `query_string` and `fragment_id` are optional and used as additional information for a server,e.g., as a local identifier within the requested document.

Example 2.1. As an example, the following URI identifies and locates a resource which describes the person Tim Berners-Lee:

`http://www.w3.org/People/Berners-Lee/`

This particular resource can be accessed with the HTTP protocol and is published by the authority `w3.org`, which represents the World Wide Web Consortium (W3), the main international standards organisation for the World Wide Web.

Furthermore, we use in this thesis the concept of a pay-level domain to indicate the authority, also referred to as data provider or publisher, of a URI.

Definition 1 (Pay-level domain (PLD))**.**
A pay-level domain (PLD) refers to a sub part of a fully qualified domain name (FQDN), the domain part of a URI [Lee et al., 2009]. It is a direct sub-domain of a top-level domain (TLD) or a second-level country domain (ccSLD).

Example 2.2. As an example, the PLD of the URI `http://www.deri.ie/` would be `deri.ie`, similarly the URI `http://www.bbc.co.uk/` has the PLD `bbc.co.uk`.

Throughout the thesis, we prefer the notion of a pay-level domain since fully qualified domain names over-exaggerate the diversity of the data: for example, sites such as `livejournal.com`, a social media platform, assign different subdomains to individual users (e.g., `danbri.livejournal.com`), leading to millions of FQDNs on one site, all under the control of one publisher. Henceforth, when we mention domain, we thus refer to a PLD (unless otherwise stated).

HTTP: HTTP is the core communication protocol for software clients to access information on the Web and to exchange and transfer data, with HTTP/1.1 being the currently used version [Fielding et al., 1999]. The communication between client and server is established by the client who sends a HTTP request and receives a response message from the server in return.

A HTTP request consists of two parts, a header and a body. The header message contains information to specify operating parameters of the connection, such as the preferred encoding, file format or to identify the client. The request body is used to send data to the server, e.g., submitting the information of a Web form, uploading a file or updating the content of a URI. There exist three core communication methods (`GET`, `POST` and `HEAD`) since version 1 with five additional methods added to version 1.1 to deal more efficiently with proxies, virtual hosts or persistent connections [Fielding et al., 1999]. The approaches investigated in this thesis make only use of the `GET` method, which, by definition, requests a representation for a URI from the specified server. For example, every HTML Web browser issues such a `GET` request to receive and display the content for the given URI.

Example 2.3. The following is an example request-line for a browser accessing our previous example URI by means of the `GET`:

```
GET http://www.w3.org/People/Berners-Lee/ HTTP/1.1
```

The retrieval mechanism that uses HTTP to obtain a copy or representation for a URI is commonly referred to as "*dereferencing a URI*".

Once a client issued a request, the server returns a response message which contains a status-line, a response header and an optional message body. The first line contains a status code indicating if the server could understand and process the request. Furthermore, we can distinguish between five categories of such status codes based on the first leading digit. Codes starting with:

1xx are purely informal,

2xx indicate that the request was successfully received and processed,

3xx inform clients that the location of the requested URI has (temporarily or permanently) changed and the response also contains the new location of the redirected resource. Especially these redirect mechanism is widely used for publishing Linked Data and to guide a client to the RDF representation of a URI.

4xx signal client side request errors, such as a request for a non existing or non-authorised source, or a malformed request in general,

5xx warn the client that a server-side error occurred, e.g., the specific request method is not supported by the server or other internal errors like timeouts.

The response header contains again meta information about the response type, such as the file format, content length or server specifications. The message body can be the requested data or other information which relates to the request, e.g., additional information if some server errors occurred.

Moreover, the usage of header fields allows to issue conditional client requests for which a server only returns data if the conditions are met. Examples for such conditions might be that a client is only interested to receive the content of a URI if changed since the last request (e.g., proxy caches) or that a client is only interested in specific file formats such as a HTML, XML or RDF. Specifically, requesting a certain representation format for a URI is a common practise for the decentralised query approaches this thesis is centred around.

MIME-TYPE AND CONTENT NEGOTIATION The representation format for the content of a URI is described by the Multipurpose Internet Mail Extensions (MIME): originally designed as a standard to describe the content of emails, e.g., to send binary attachments or specific text encodings. These MIME types, also called a content type, are widely used to identify file formats on the Web and consists of two parts separated by a "/". The first part specifies the communication medium (e.g., image, audio, text, application) and the second part the specific format (e.g., pdf, jpeg or xml). For example, the MIME type for HTML is *text/html*. A list of known content types for the Web is managed by the Internet Assigned Numbers Authority (IANA) [1].

A Web server can potentially offer several different representations formats for the content of a URI, which in itself, only identifies and locates a given resource. Thus, the HTTP protocol allows to specify the required or desired representation formats of a URI with the request header. Naturally, a server tries to serve the content type as requested by the client. It is also possible to name several content-types with their relative degree of preference. This process is referred to as *content-negotiation* – a process to negotiate the best possible presentation of a URI for a client. This allows to design specialised clients, such as Web browsers which render HTML. Other examples are Linked Data browsers or query engines which require that the returned content is served as RDF.

Example 2.4. An example for such a selection process is:

```
GET "Accept: application/rdf+xml; text/html;" http://www.w3.
org/People/Berners-Lee/ HTTP/1.1
```

This GET method requests that the `w3.org` server returns the content for the specific URI as RDF (`application/rdf+xml`) and if not available as an HTML document (`text/html`).

The architecture of the Web and the tremendous effort of the W3C to establish standards, such as HTTP and URI, transformed the Web into an open and globally accessible distributed hypermedia system which allows users to publish, link and interact with documents containing information in different formats, such as text, images, videos and audio. As a result, the Web can be classified as a self-

1 http://www.iana.org/

organising, *technical autopoietic system*[2] with evidential properties such as (i) the lack of a central controlling agency, (ii) the integration of use and evolution, (iii) inner dynamics, and partially autonomous processes [Andersen, 1998]. We can observe that many diverse and heterogenous units participate on the Web in a parallel, uncontrolled and distributed manner by creating, publishing, consuming and organising information. The combination between human and legal agents and also the physical hardware even allows to classify of the Web as a self-organising sociotechnological system [Fuchs, 2005]. Undeniably, the Web is highly dynamic in its very nature but also shows emerging and stable patterns.

Lee et al. [2009] reported that over 90% of Web documents contained HTML content in 2007. That implies that the sheer amount of available and published information are unstructured text created by and for humans. Furthermore, the increasing size of the Web – currently estimated to be at least 8 billion documents [de Kunder, 2012] – demands the use of machines to efficiently collect, organise and manage information.

However, the amount of unstructured information in the form of text introduces (i) irregularities (e.g., different data formats) and (ii) ambiguities (e.g., is Mercury the element, the plant or the god?) during information extraction processes [Nadeau and Sekine, 2007], which are necessary to develop efficient search & query components. Moreover, the existence of different human languages, domain specific or social driven vocabularies (e.g., slang) and the diverse backgrounds of people increase the complexity and dimension of automatically processing and understanding the words and information in text [Buitelaar and Cimiano, 2008]. In addition, it is far from trivial to extract structured facts from unstructured text, such as the address or descriptions of a person [Arasu and Garcia-Molina, 2003]. Thus, the large amount of unstructured text prohibits current search engines from answering arbitrarily structured queries.

Example 2.5. For example, current free text search engines fail to answer a query like "*What are the current temperatures in the capitals of Europe?*". To answer such a query, a search engine has to know (i) that a free-text term uniquely identifies a city and does not refer to other synonymic concepts (e.g., such as the term "Boston" can refer to a city but also to a band), (ii) that cities can be capitals of countries, (iii) that countries are associated with continents such as Europa, and (iv) that a certain string representation is a temperature value. Eventually, the systems must connect and relate all of this information, to generate a final result set.

One major problem for search & query components is that the traditional Web does not express enough of its information in a machine readable format.

2.2 THE SEMANTIC WEB AND LINKED DATA

The shortcomings of the traditional Web with it vast amount of unstructured information lead to the idea of Berners-Lee for a machine readable Web, in his roadmap towards the so called "*Semantic Web*" [Berners-Lee, 1998]. The Semantic Web is an extension of the existing Web, which transforms a mostly human-readable "*Web of Documents*" to a "*Web of Data*" which can be processed by software agents. Its fundamental technology is the W3C's Resource Description Framework (RDF) [Lassila and Swick, 1999]; a data model to represent and interchange information on the Web and ease the integration and connection of data elements across domains.

2 An autopoietic system is an environment which is organised by its processes rather than its elements.

2.2.1 RDF

RDF was original designed to describe meta data about Web resources, but is now more generally used to model information about any form of *resource* through RDF statements. It is a form of highly-normalised relational model with binary relations between global unique identifiers [Decker et al., 2000].

An RDF statement is a tuple consisting of three elements, so called RDF resources. An RDF resource can be either a Web resource (URI) or a real-world object which again can be identified by a URI or a string value, called a literal[3], e.g., `"Dublin"`. In the case that a node is not identified with a URI or a literal, it is represented as an anonymous or unnamed resources called a blank node with the notation `_:name`. An RDF statement is also called a and is expressed as a tuple of the form:

```
(subject, predicate, object)
```

We now formally define RDF (cf. [Hayes and McBride, 2004]) using standard notation and formal definitions as follows:

Definition 2 (RDF Term, Triple and Graph)**.**
The set of RDF terms *consists of the set of URIs* $\mathbf{U}$*, the set of blank-nodes* $\mathbf{B}$ *and the set of literals* $\mathbf{L}$ *(which includes plain and datatype literals).* $\mathbf{U}$*,* $\mathbf{B}$ *and* $\mathbf{L}$ *are pairwise disjoint. An* RDF triple $\mathsf{t} \coloneqq (\mathsf{s}, \mathsf{p}, \mathsf{o})$ *is an element of the set* $\mathbf{G} \coloneqq \mathbf{UB} \times \mathbf{U} \times \mathbf{UBL}$ *(where, e.g.,* $\mathbf{UB}$ *is a shortcut for set-union). Here* s *is called subject,* p *predicate, and* o *object. A finite set of RDF triples* $\mathsf{G} \subset \mathbf{G}$ *is called an* RDF graph*. We use the functions* $\mathsf{subj}(\mathsf{G})$, $\mathsf{pred}(\mathsf{G})$, $\mathsf{obj}(\mathsf{G})$, $\mathsf{terms}(\mathsf{G})$*, to denote the set of all terms projected from the subject, predicate, object and any position of a triple* $\mathsf{t} \in \mathsf{G}$ *respectively.*

REPRESENTATION FORMATS Most RDF syntaxes allow to use Compact URI (CURIE) names [Birbeck and McCarron, 2008] of the form `prefix:reference` to denote URIs. For example, a parser expands the CURIE `foaf:name` in combination with the syntactic definition of the namespace prefix to the full URI `http://xmlns.com/foaf/0.1/name`. Prefixes for the abbreviated CURIE names used throughout the thesis are available in Appendix A.

RDF data is commonly serialised either as RDF/XML [Swartz, 2004] or as Terse RDF Triple Language (Turtle) [Beckett and Berners-Lee, 2008]. The MIME types are `application/rdf+xml` [Swartz, 2004] and `text/turtle`. In general, the Turtle syntax is seen as more human friendly than RDF/XML. As such, we may use Turtle syntax throughout this thesis.

In the Turtle syntax, RDF statements are written as whitespace separated triples and graphs are written as '.'-separated lists of such triples. Brackets (<>) denote URIs and quotes ("") denote literals. Blank node identifiers start with '_ :'.

Example 2.6. For example, the following data shows two statements about the city Dublin (`dbpedia:Dublin`) in the Turtle syntax.

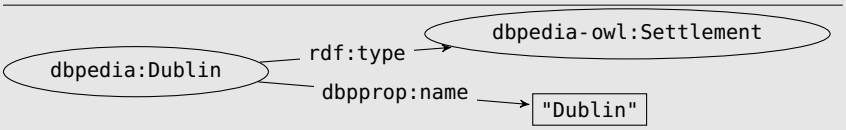
```
dbpedia:Dublin rdf:type dbpedia-owl:Settlement;
               dbpprop:name "Dublin" .
```

The first triple states that the resource `dbpedia:Dublin` is a "settlement" and the second provides a human readable label for the resource.

3 Literals may not be used as subjects or predicates in RDF statements, Blank nodes may not be used as predicates.

An RDF triple can also be graphically represented as an edge between two nodes in a directed labelled multi graph, where the predicate is the edge between the subject and object nodes of the triple. In the graph representation, we draw an RDF triple as two labelled nodes (subject and object) connected with a directed labelled edge (predicate). Note, one problem with the directed labelled graph representation of RDF is that a predicate of a triple (an edge label) can also occur as the subject or object of another triple, which would require to either duplicate resources or allow connections to edges. As such, Hayes and Gutiérrez [2004] proposed to model RDF graphs as bipartite graphs, which are well-known mathematical objects and have a formal representation. This would allow to apply techniques and results of graph theory, and also to use generic graph algorithms and visualisation libraries. However, we use the graph model only to visualise the RDF data examples in this thesis and believe that the directed labelled graphs serves as a good and intuitive representation.

Example 2.7. In the following, we visualise the above two statements as a directed graph, where we denoted URIs with ellipses and literals with squares.



RDF ON THE WEB Early RDF data published on the Web was mainly meta data about Web resources, as initially designed (e.g. RSS 1.0 [RSS, 1999]). Most of the RDF was published as data dumps in isolated documents (mostly) missing important links to other RDF documents. This already added some structured data to the Web but was far from transforming the Web of Documents into a Web of (interlinked) Data. The missing links between the RDF sources prevented the navigation to or discovery of new information as users (with browsers) or agents (with Web crawlers) are able to do in the traditional Web by following links. At that time, Semantic Web applications, such as search engines, had to know the locations (URIs) of the RDF dumps to harvest and merge the contents and locally build the desired global graph. The missing links prevented software agents to start form at least one known URI and automatically follow links to discover more structured information.

However, some domains already provided these links between the RDF documents. One example of a successful domain is the Friend Of A Friend (FOAF) project, which publishes *interlinked* data on the Web. The fundamental idea of FOAF is to build a distributed decentralised social network, published as RDF on the Web. Users can enter this FOAF Web by any identifier of a person and access any relevant information, such as the name, gender or general interest of that person. In addition, users can navigate the social network of a person by following links which identify that two people know each other. In that way, applications are able to understand, browse and download the social network of people by following typed links.

2.2.2 Linked Data

In 2006, Berners-Lee [2006] addressed the problem that most of the published RDF data on the Web was either available as large data dumps or in isolated documents. He introduced the “Linked Data” design principles to create a Web of interlinked

data. A set of four simple guidelines promote the tight integration of the RDF graph model with the architecture of the World Wide Web to create a scaleable network of interlinked data which can be consumed and browsed by machines and humans alike. The core of these principles is to use dereferenceable HTTP URIs as real-world identifiers and the necessary requirement to link to other URIs, preferable with the RDF model. In detail, the four principles, as defined by Berners-Lee [2006], are:

LDP1: Use URIs to unambiguously identify/refer to (real-world) *things*.

LDP2: Use HTTP URIs so that legal and software agents can deference those identifiers.

LDP3: Upon requests, provide useful information using the standards, preferable RDF and SPARQL (which we introduce later).

LDP4: Include links using externally dereferenceable URIs.

In short, these principles shall ensure that RDF data published on the Web contain HTTP-dereferenceable URIs which return more information about the respective URIs upon lookup (possibly also from third-party data providers).

The following introduces our real-world Linked Data example, which we also use to exemplify definitions, algorithms and the general idea of our approaches throughout the thesis.

Example 2.8. We illustrate in Figure 2.1 an RDF (sub-)graph extracted from five real-world interlinked documents on the Web of Data.[a]

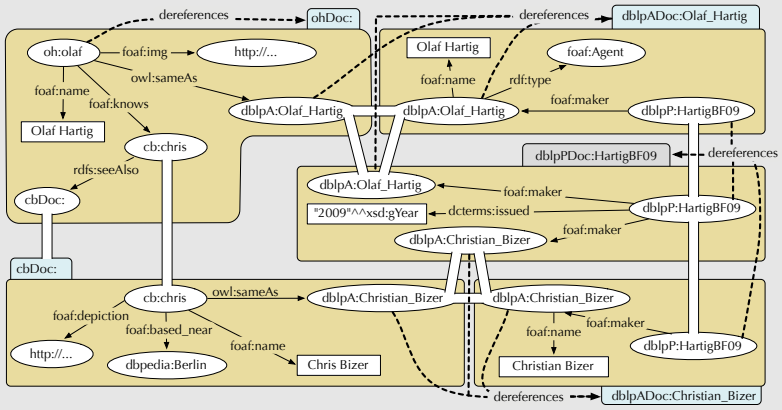


Figure 2.1: Snapshot of a subgraph of five documents from the Web of Data. Individual documents are associated with individual background panes. The URI of each document is attached to its pane with a shaded tab. The same resources appearing in different documents are joined using "bridges". Links from URIs to the documents they dereference to are denoted with dashed links. RDF triples are denoted following the aforementioned conventions within their respective document.

The graph models information about two real-world persons and a paper that they coauthored together. These real-world elements are identified with HTTP URIs as per **LDP1** and **LDP2**. One of the authors is identified by the two URIs `oh:olaf` and `dblpA:Olaf_Hartig`and the other by the two identifiers `cb:chris` and `dblpA:Christian_Bizer`. The URI `dblpP:HartigBF09` refers to the publication both authors share.

These five resources are mentioned and described in at least one of the five documents:

`ohdoc:`, `cbdoc:` refer to the personal FOAF profile documents that each author created for themselves;

`dblpadoc:olaf`..., `dblpadoc:chris`... refer to information exported from the "DBLP Computer Science Bibliography"[b] for each author, including a publication list;

`dblppdoc:hartigbf09` provides information about the co-authored paper exported from DBLP.

Each of these documents are available as RDF/XML on the Web and can be retrieved by performing a HTTP `Get` operation on their URIs. Furthermore, we indicated with the dashed lines which URIs dereference to which sources, according to the principle **LDP3**. For example, a agent that performs a HTTP `Get` on `oh:olaf` will be redirected to the source URI `ohDoc:`. Furthermore, some of the sources link to other sources by reusing existing dereferenceable URIs (**LDP4**). We indicate the reuse of URIs with white connectors between the nodes. For example, the URI `dblpA:Olaf_Hartig` in the top right source is reused in the top left source to indicate that the URI `oh:olaf` identifies the same person as the reused URI.

a As last accessed on 2012-06-23.
b `http://www.informatik.uni-trier.de/~ley/db/`

We now provide some notations which helps to formalise the four principles and relate them to RDF and HTTP. As per [Hartig et al., 2009], we do not consider temporal issues with, e.g., HTTP-level functions.

Definition 3 (Data Source and Linked Dataset)**.**
We define the http-download *function* $\texttt{get} : \mathbf{U} \rightarrow 2^{\mathbf{G}}$ *as the mapping from URIs to RDF graphs provided by means of HTTP lookups which directly return status code* `200OK` *and data in a suitable RDF format. We define the set of (RDF)* data sources $\mathbf{S} \subset \mathbf{U}$ *as the set of URIs* $\mathbf{S} := \{s \in \mathbf{U} : \texttt{get}(s) \neq \emptyset\}$*. We define a Linked Dataset as* $\Gamma \subset \texttt{get}$*; i.e., a finite set of pairs* $(s, \texttt{get}(s))$ *such that* $s \in \mathbf{S}$*. The "global" RDF graph presented by a Linked Dataset is denoted as*

$$\texttt{merge}(\Gamma) := \biguplus_{(u,G)\in\Gamma} G$$

where the operator '$\uplus$*' denotes the RDF merge of RDF graphs: a set union where blank nodes are rewritten to ensure that no two input graphs contain the same blank node label [Hayes and McBride, 2004].*

Example 2.9. Taking Figure 2.1, the function call `get(ohDoc:)` = { (`oh:olaf:`, `foaf:name`, `"Olaf Hartig"`), ...} gives an RDF graph containing the five triples in that document. However, `get(oh:olaf)` = $\emptyset$ since it does not return a `200 Okay` (redirects are supported in the next step). Thus, `ohDoc:` $\in \mathbf{S}$ whereas `oh:olaf` $\notin \mathbf{S}$. If we denote Figure 2.1 as the Linked Dataset Γ, we can say that $\Gamma = \{(\texttt{ohDoc:}, \texttt{get}(\texttt{ohDoc:}), ...\}$, contains five (*URI, RDF-graph*) pairs. Then, `merge`(Γ) is the set of all 17 RDF triples shown in Figure 2.1.

Definition 4 (Dereferencing RDF)**.**
A URI may issue a HTTP redirect to another URI with a `30x` *response code, with the*

target URI listed in the `Location:` *field of the HTTP header. We model this redirection function as* $\mathsf{redir} : \mathbf{U} \to \mathbf{U}$*, which first strips the fragment identifier of a URI (if present) and would then map a URI to its redirect target or to itself in the case of failure (e.g., where no redirect exists). We denote the fix-point of* redir *as* redirs*, denoting traversal of a number of redirects (a limit may be imposed to avoid cycles). We denote* dereferencing *by the composition* $\mathsf{deref} \coloneqq \mathsf{get} \circ \mathsf{redirs}$*, which maps a URI to an RDF graph retrieved with status code* `200OK` after *following redirects, or which maps a URI to the empty set in the case of failure. We denote the set of* dereferenceable URIs *as* $\mathbf{D} \coloneqq \{\mathsf{d} \in \mathbf{U} : \mathsf{deref}(\mathsf{d}) \neq \emptyset\}$*; note that* $\mathbf{S} \subset \mathbf{D}$ *and we place no expectations on what* $\mathsf{deref}(\mathsf{d})$ *returns, other than returning* some *valid RDF. As a shortcut, we denote by* $\mathsf{derefs} : 2^{\mathbf{U}} \to \mathbf{U} \times 2^{\mathbf{G}}; \mathsf{U} \mapsto \{(\mathsf{redirs}(\mathsf{u}), \mathsf{deref}(\mathsf{u})) \mid \mathsf{u} \in \mathsf{U} \cap \mathbf{D})\}$ *the mapping from a set of URIs to the Linked Dataset it represents by dereferencing all URIs (only including those in* $\mathbf{D}$ *which return some RDF).*

Example 2.10. Taking Figure 2.1, dereferenceable relationships between resources and documents are highlighted with the dashed lines. Excluding `cbDoc:` (which must be looked up directly), the other four documents can be retrieved by dereferencing the URI of their main resource; for example, dereferencing `oh:olaf` over HTTP returns the document `ohDoc:` describing said resource: `oh:olaf` redirects to `ohDoc:`, denoted $\mathsf{redir}(\texttt{oh:olaf}) = \texttt{ohDoc:}$. No further redirects are possible, and thus $\mathsf{redirs}(\texttt{oh:olaf}) = \texttt{ohDoc:}$. Dereferencing `oh:olaf` gives the RDF graph in the document `ohDoc:`, where $\mathsf{deref}(\texttt{oh:olaf}) = \mathsf{get}\big(\mathsf{redirs}(\texttt{oh:olaf})\big) = \mathsf{get}(\texttt{ohDoc:})$. Instead taking the URI `cb:chris`, $\mathsf{redir}(\texttt{cb:chris}) = \texttt{cb:chris}$ and $\mathsf{get}(\texttt{cb:chris}) = \emptyset$; this URI is not dereferenceable. Thus we can say that $\texttt{oh:olaf} \in \mathbf{D}$ and $\texttt{ohDoc:} \in \mathbf{D}$ whereas $\texttt{cb:chris} \notin \mathbf{D}$.

As a result of the work of the Linked Data community, we can witness how organisations, companies, governments, librarians and various research communities participate to the growth and wealth of to the Web of interlinked Data [Bizer et al., 2009]. Just to name some examples, news portal and multimedia domains, such as the New York Times or BBC, publish their meta-data about articles and programs according to the principles of Linked Data. The DBpedia project provides a Linked Data version of the Wikipedia encyclopaedia and is one of the most broadly linked datasets available [Auer et al., 2007]. Content management systems like Drupal 7[4] and knowledge management system like SemWiki[5] are also part of the Linked Data ecosystem.

2.2.3 RDFS & OWL

RDF allows to define resources as members of classes, which themselves are described as RDF resources with the RDF predicate `rdf:type`:

Example 2.11. For example, the following statement states the fact that a resource is a person:

```
cb:chris rdf:type foaf:Person
```

The RDF graph model can be used to create vocabularies which represent information of any real-world domain in the form of statements about resources.

4 http://drupal.org/node/725382
5 http://km.aifb.kit.edu/ws/semwiki2006/

The typing and inheritance of classes and RDF predicates in and across vocabularies, also called properties, can be described with the RDF Vocabulary Description Language: RDF Schema (RDFS) [Brickley and Guha, 2004], which was designed to represent vocabularies on the Web. RDFS allows primarily to define classes and properties and to describe the relationships between them.

Example 2.12. As an example, we show a subset of RDFS definitions in a "schema document" extracted from the real-world FOAF ontology in Figure 2.2.

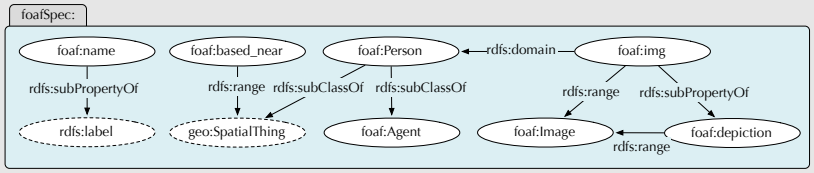


Figure 2.2: Snapshot of an example schema document from the Web of Data, taken from the Friend Of A Friend (FOAF) Ontology. External terms are represented in ellipses with dashed lines.

Although we leave it implicit, all terms in the `foaf:` namespace (including the predicates and values for `rdf:type` represented in Figure 2.1) dereference to `foafSpec:`. The relations between classes and properties shown in this document are well defined (using model-theoretic semantics) by the RDFS standard [Hayes and McBride, 2004].

There exists four core terms in RDFS to specify the relations between classes and properties, namely `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range`. The first term allows to define the relationship between classes, it enables to specify that the instances of one class are also instances of another class. Similarly, `rdfs:subPropertyOf` describes the hierarchy of properties. The last two terms (`rdfs:domain` and `rdfs:range`) are used to state that the subject of a given property is a member of a class, or that the object value of a property is a member of a given class, respectively.

Example 2.13. For example, the FOAF vocabulary (cf. Figure 2.2) states that members of `foaf:Person` are also members of `foaf:Agent` and that the property `foaf:name` is a sub-property of `rdfs:label`. The FOAF vocabulary also uses the other two prominent RDFS property and encodes that resources that appear in the subject position of `foaf:img` are `foaf:Person`'s and the resources in the object position of such statements are instances of the class `foaf:Image`.

In addition, to describe more complex relations and domains, the Web Ontology Language (OWL) extends RDFS [Antoniou et al., 2003; Dean and Schreiber, 2004] and allows to model more expressive ontologies for certain domains. Machines can use these vocabularies/ontologies to infer additional information based on explicit data, which allows for better integration and interoperability of data among descriptive communities.

We support a small subset of OWL 2 RL/RDF rules, given in Table 2.1, which constitute a partial axiomatisation of the OWL RDF-Based Semantics. The RDFS rules used in this thesis are the subset of the ρDF rules proposed by Muñoz et al. [2009] that deal with instance data entailments (as opposed to schema-level entailments

of triples in the RDFS vocabulary). [6] Moreover, these rules are the most used rules for the available Linked Data on the Web, as a recent published survey by Glimm et al. [2012] shows. The authors studied the use of RDFS and OWL features in a large crawl of the Web of Data (viz., the Billion Triple Challenge Dataset (BTC'11), which we detail in the next chapter). They apply PageRank [Page et al., 1999] over the documents contained within the dataset and summating the rank of all documents using each feature. They found that RDF(S) features were the most prominently used. Respectively from features ranked 1–6: `rdfs:Property`, `rdfs:range`, `rdfs:domain`, `rdfs:subClassOf`, `rdfs:Class`, `rdfs:subPropertyOf`.

Considering the most used OWL features on the Web of Data, Glimm et al. [2012] found that the `owl:sameAs` property occurred most frequently, though other features of OWL like `owl:FunctionalProperty` had a higher rank: the most broadly linked (and thus highly ranked) documents in the Web of Data are vocabularies, not the "data-level documents" in which `owl:sameAs` relations frequently appear. As such, we chosen to support the semantics of equality (particularly replacement) for `owl:sameAs` in this thesis. Note that these rules support the RDFS/OWL features originally recommended for use by Bizer et al. [2007] when publishing Linked Data. The rules we consider are given in Table 2.1. More recent guidelines [Heath and Bizer, 2011] recommend use of additional OWL features, however, we leave support for more expressive OWL reasoning to future work.

`owl:sameas` The `owl:sameAs` relation states that two URIs identify the same real-world element. In general, this involves the reuse of external URI (see **LDP4**) which helps to create important links between sources (as we can see in our example Figure 2.1).

Example 2.14. Taking Figure 2.1 and the following statements of one the sources (`ohDoc:`):

```
oh:olaf owl:sameAs dblpA:Olaf_Hartig
```

The data states that the two URIs `oh:olaf` and `dblpA:Olaf_Hartig` refer to the same real-world person. Furthermore, the `owl:sameAs` statement creates the important link between the two documents `ohDoc:` and `dblpADoc:Olaf_Hartig` by reusing the person URI `dblpA:Olaf_Hartig`.

Interestingly, various authors have looked specifically at the usage and quality of use of these `owl:sameAs` statements on the Web of Data [Ding et al., 2010; Halpin et al., 2010; Hogan et al., 2012b]. Halpin et al. [2010] look at the semantics and quality of `owl:sameAs` links in Linked Data; manually inspecting five hundred `owl:sameAs` sampled from the Web of Data; they found that judges often disagreed on what resources should be considered the same and what resources shouldn't. They estimated an accuracy for `owl:sameAs` links – where sameness could be confidently asserted for the sampled relations – at around 51% ($\pm$21%). We performed a similar analysis on one thousand `owl:sameAs` relations, where we asked a different question – is there anything between these two resources to confirm that they are not the same? – and where we estimated a estimated precision of 97.2% [Hogan et al., 2012b]. This analysis was applied over pairs sampled from the *closure* of same-as relations. During our analysis, we found many pairs of resources for which very little knowledge was locally or externally available (for one or both). If there was no information to suggest that they were *not* the same, we would give this pair the

6 We drop *implicit typing* [Muñoz et al., 2009] rules as we allow generalised RDF in intermediate inferences.

	ID	Body	Head
RDFS	PRP-SPO1	?p$_1$ rdfs:subPropertyOf ?p$_2$. ?s ?p$_1$?o.	?s ?p$_2$?o.
	PRP-DOM	?p rdfs:domain ?c . ?s ?p ?o .	?p a ?c.
	PRP-RNG	?p rdfs:range ?c . ?s ?p ?o .	?o a ?c .
	CAX-SCO	?c$_1$ rdfs:subClassOf ?c$_2$. ?s a ?c$_1$.	?s a ?c$_2$.
Same-As	EQ-SYM	?x owl:sameAs ?y .	?y owl:sameAs ?x .
	EQ-TRANS	?x owl:sameAs ?y . ?y owl:sameAs ?z .	?x owl:sameAs ?z .
	EQ-REP-S	?s owl:sameAs ?s . ?s ?p ?o .	?s′ ?p ?o .
	EQ-REP-P	?p owl:sameAs ?p . ?s ?p ?o .	?s ?p ?o .
	EQ-REP-O	?o owl:sameAs ?o . ?s ?p ?o .	?s ?p ?o .

Table 2.1: RDFS (ρDF subset) and owl:sameAs (OWL 2 RL/RDF subset) rules

"benefit of the doubt", taking the perspective that merging (aka. consolidating) the resources would not cause any notable data issues. There are other studies which question the quality of such statement [Halpin et al., 2010; Hogan et al., 2012b]. However, for this work at hand we rely on the published studies and assume the correctness of owl:sameAs information found on the Web.

We also provide some definitions regarding the rule-based inferencing based on the our ruleset (cf.tbl:rules) we will use in the following of this thesis.

Definition 5 (Entailment Rules & Least Model)**.**
An entailment rule is a pair $r = (\mathtt{Body}, \mathtt{Head})$ *(cf. Table 2.1) such that* $\mathtt{Body}$, $\mathtt{Head} \subset \mathbf{Q}$*; and* $\mathsf{vars}(\mathtt{Head}) \subseteq \mathsf{vars}(\mathtt{Body})$*. The immediate consequences of* r *for a Linked Dataset* Γ *are denoted and given as:*

$$\mathfrak{T}_r(\Gamma) \quad := \{\mu(\mathtt{Head}) \mid \mu \in [\![\mathtt{Body}]\!]_\Gamma\} \setminus \mathsf{merge}(\Gamma)\,.$$

In other words, $\mathfrak{T}_r(\Gamma)$ *denotes the direct unique inferences from a single application of a rule* r *against the merge of RDF data contained in* Γ*. Let* R *denote a finite set of entailment rules. The immediate consequences of* R *over* Γ *are given analogously as:*

$$\mathfrak{T}_R(\Gamma) \quad := \bigcup_{r \in R} \mathfrak{T}_r(\Gamma)\,.$$

This is the union of a single application of all rules in R *over the data applied to the (raw) data in* Γ*. Furthermore, let* $v \in \mathbf{U}$ *denote a fresh URI which names the graph* G^R *of data inferred by* R*, and let* $G^R_0 = \emptyset$*. Now, for* $i \in \mathbb{N}$*, define:*

$$\begin{aligned} \Gamma^R_i &:= \Gamma \cup \left\{\left(v, G^R_i\right)\right\} \\ G^R_{i+1} &:= \mathfrak{T}_R(\Gamma^R_i) \cup G^R_i \end{aligned}$$

The least model *of* Γ *with respect to* R *is* Γ^R_n *for the least* n *such that* $\Gamma^R_n = \Gamma^R_{n+1}$*; at this stage the closure is reached and nothing new can be inferred.*[7] *Henceforth, we denote this least model with* $\Gamma \bullet R$.

Example 2.15. Let R denote the set of rules in Table 2.1. Also, consider Γ as the Linked Dataset comprising of $\left(\mathtt{ohDoc:}, \mathsf{get}(\mathtt{ohDoc:})\right)$ from Figure 2.1 and a second named graph called foafSpec: with the following subset of triples from Figure 2.2:

7 Since our rules are a syntactic subset of Datalog, there is a unique and finite least model (assuming finite inputs).

```
foaf:img rdfs:domain foaf:Person ;
rdfs:range foaf:Image ;
rdfs:subPropertyOf foaf:depiction .
foaf:Person rdfs:subClassOf foaf:Agent .
```

These (real-world) triples can be retrieved by dereferencing a FOAF term; *e.g.*, deref(`foaf:img`). Now, given Γ and R, then $G^R_0 = \emptyset$, $G^R_1 = G^R_0 \cup \mathfrak{T}_R(\Gamma^R_0)$ where, by applying each rule in R over Γ once, $\mathfrak{T}_R(\Gamma^R_0)$ contains the following triples (abbreviating URIs slightly):

```
oh:olaf foaf:depiction <http...> .                    #PRP-SPO1
oh:olaf a foaf:Person .                               #PRP-DOM
<http...> a foaf:Image .                              #PRP-RNG
dblpA:Olaf owl:sameAs oh:olaf .                       #EQ-SYM
dblpA:Olaf foaf:knows cb:chris .                      #EQ-REP-S
...
```

Subsequently, $\Gamma^R_1 = \Gamma \cup \{(\upsilon, G^R_1)\}$, where υ is any built-in URI used to identify the graph of inferences and where G^R_1 contains the unique inferences thus far (listed above). Thereafter, $G^R_2 = G^R_1 \cup \mathfrak{T}_R(\Gamma^R_1)$, where $\mathfrak{T}_R(\Gamma^R_1)$ contains:

```
oh:olaf a foaf:Agent .                                #CAX-SCO
dblpA:Olaf foaf:depiction <http...> .                 #EQ-REP-S
dblpA:Olaf a foaf:Person .                            #EQ-REP-S
dblpA:Olaf owl:sameAs dblpA:Olaf .                    #EQ-REP-S
oh:olaf owl:sameAs oh:olaf .                          #EQ-REP-S
...
```

As before, $\Gamma^R_2 = \Gamma \cup \{(\upsilon, G^R_2)\}$, where G^R_2 contains all inferences collected thus far, and $G^R_3 = G^R_2 \cup \mathfrak{T}_R(\Gamma^R_2)$, where $\mathfrak{T}_R(\Gamma^R_2)$ contains:

```
dblpA:Olaf a foaf:Agent .                             #CAX-SCO
```

This is then the closure since $\mathfrak{T}_R(\Gamma^R_3) = \emptyset$; nothing new can be inferred, and so $\Gamma^R_3 = \Gamma^R_4$. And thus we can say that $\Gamma \bullet R = \Gamma^R_3 = \Gamma \cup (\upsilon, G^R_3)$.

Finally, we highlight that the openness of the Web poses some challenges for the reasoning process over Web data. Aside from pure efficiency and scalability concerns, the freedom associated with publishing on the Web – where anyone can say anything (almost) anywhere – causes significant obstacles with respect to the trustworthiness of data when performing automated inferencing. On a schema level, for example, various obscure documents on the Web of Data make nonsensical definitions that would (naïvely) affect reasoning across all other documents [Bonatti et al., 2011]. Various authors have proposed mechanisms to incorporate notions of provenance for schema data into the inferencing process. One such procedure, called *authoritative reasoning*, only considers the schema definitions for a class or property term that are given in its respectively dereferenceable document [Bonatti et al., 2011; Cheng and Qu, 2008; Hogan et al., 2009]. We will use in Chapter 4 authoritative reasoning in our approach to avoid the unwanted effects of third-party schema contributions during RDFS reasoning. Delbru et al. [2011] propose an alternative solution called *context-dependent reasoning* (or quarantined reasoning), where a closed scope is defined for each document being reasoned over, incorporating the document itself and (recursively) other documents it imports or links. Thus, obscure third party documents cannot inject unwanted schema into the in-

ferencing process since they fall outside the quarantined scope. We use in Chapter 4 a similar import mechanism to dynamically collect the schema data from the Web.

2.2.4 SPARQL

We now introduce some of the core concepts and notation for the RDF query language SPARQL [Pérez et al., 2009; Prud'hommeaux and Seaborne, 2008]. SPARQL is a protocol and query language for RDF and enables to retrieve and manipulate data. A query can consist of triple patterns, conjunctions, disjunctions, and optional patterns [Prud'hommeaux and Seaborne, 2008] It contains two parts, a query type and the `WHERE` clause defining the query patterns. There exist four query types, namely, `SELECT` to extract raw data, `CONSTRUCT` to return an RDF graph from the obtained query results, `ASK` to indicate with a boolean value if the query can be answered or not, and `DESCRIBE` which delivers an RDF graph that describes the results.

We now provide some preliminaries.

Definition 6 (Variables, Triple Patterns & BGPs)**.**
Let $\mathbf{V}$ *be the set of variables ranging over* $\mathbf{UBL}$*. A* triple pattern $\mathrm{tp} \coloneqq (\mathrm{s}, \mathrm{p}, \mathrm{o})$ *is an element of the set* $\mathbf{Q} \coloneqq \mathbf{VUL} \times \mathbf{VU} \times \mathbf{VUL}$*. For simplicity, we do not consider blank-nodes in triple patterns (they could be roughly emulated by an injective mapping from* $\mathbf{B}$ *to* $\mathbf{V}$*). A finite (herein, non-empty) set of triple patterns* $\mathrm{Q} \subset \mathbf{Q}$ *is called a* Basic Graph Pattern*, or herein, simply a* query*. We use* $\mathsf{vars}(\mathrm{Q}) \subset \mathbf{V}$ *to denote the set of variables in* Q*. Finally, we may overload graph notation for queries, where, e.g.,* $\mathsf{terms}(\mathrm{Q})$ *returns all elements of* $\mathbf{VUL}$ *in* Q*.*

Definition 7 (SPARQL solutions)**.**
A partial function $\mu : \mathrm{dom}(\mu) \cup \mathbf{UL} \to \mathbf{UBL}$ *is a* solution mapping *with a domain* $\mathrm{dom}(\mu) \subset \mathbf{V}$*. A solution mapping binds variables in* $\mathrm{dom}(\mu)$ *to* $\mathbf{UBL}$ *and is the identify function for* $\mathbf{UL}$*. Overloading notation, let* $\mu : \mathbf{Q} \to \mathbf{G}$ *and* $\mu : 2^{\mathbf{Q}} \to 2^{\mathbf{G}}$ *also resp. denote a solution mapping from triple patterns to RDF triples, and basic graph patterns to RDF graphs such that* $\mu(\mathrm{tp}) \coloneqq (\mu(\mathrm{s}), \mu(\mathrm{p}), \mu(\mathrm{o}))$ *and* $\mu(\mathrm{Q}) \coloneqq \{\mu(\mathrm{tp}) \mid \mathrm{tp} \in \mathrm{Q}\}$*. We now define the set of* SPARQL solutions *for a query* Q *over a (Linked) Dataset* Γ *as*

$$[\![\mathrm{Q}]\!]_\Gamma \coloneqq \{\mu \mid \mu(\mathrm{Q}) \subseteq \mathsf{merge}(\Gamma) \wedge \mathrm{dom}(\mu) = \mathsf{vars}(\mathrm{Q})\}\,.$$

For brevity, and unlike SPARQL, solutions are herein given as sets (not multi-sets), implying a default `DISTINCT` *semantics for queries, and we assume that answers are given over the default graph consisting of the merge of RDF graphs in the dataset.*

The approaches in this thesis focus on evaluating simple, conjunctive, *basic graph patterns* (BGPs) in the `WHERE` clause. Although supported by our implementations, we do not formally consider more expressive parts of the SPARQL language, which – with the exception of `OPTIONAL` patterns in the original SPARQL specification and the patterns `MINUS`/(`NOT`) `EXISTS` defined in SPARQL 1.1 which assume a closed dataset – can be layered on top [Pérez et al., 2009].

Example 2.16. Again taking Γ from Figure 2.1, if we let Q be Query 2.1 as follows:

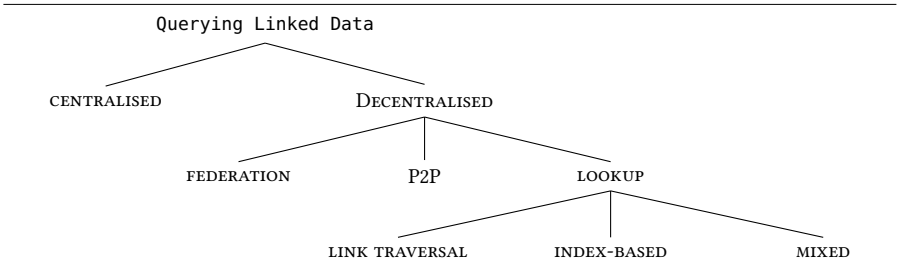


Figure 2.3: Classification of Linked Data query approaches.

```
SELECT ?maker ?issued
WHERE {
  dblpP:HartigBF09 foaf:maker ?maker ;
              dcterms:issued ?issued .
}
```

Query 2.1: Authors and creation date of a paper

Then $[\![Q]\!]_\Gamma$ would be:

?maker	?issued
dblpA:Christian_Bizer	"2009"^^xsd:gYear
dblpA:Olaf_Hartig	"2009"^^xsd:gYear

2.3 QUERYING LINKED DATA

The Web of Data can be seen as an powerful extension of the traditional Web. The consistent use of RDF for publishing information and its standardised query language SPARQL [Prud'hommeaux and Seaborne, 2008] bear the potential to turn the Web into the long envisioned decentralised and global knowledge base to which every software and legal agent can get access to consume data, query for information, but also to share, manipulate or create data [Berners-Lee, 1998; Hausenblas and Karnstedt, 2010]. For the achievement of this desirable goal, it is even more important to research and design efficient query processing engines which can operate over this decentralised and distributed collection of information.

SPARQL is originally designed to query over one single dataset rather than over several small datasets, as it is the case for Linked Data. As such, SPARQL query engines need to integrate data from various sources to evaluate a given query. A very comprehensive and detailed overview about existing solutions and concepts to query Linked Data was recently published by Hose et al. [2011]. The authors classified existing query approaches as original proposed by Domenig and Dittrich [1999] to distinguish the techniques according to how data from several sources is integrated for query processing. We similarly, categorise relevant query approaches for this thesis as depict in Figure 2.3.

The most prevalent query and search systems for RDF data apply a CENTRALISED approach, which collects data from known sources in advance and pre-processes and indexes the combined data in a centralised store; queries are evaluated using the local store. In addition, these materialised systems assume full sovereignty over the data.

Another class of query approaches, enjoying more attention recently, are DECENTRALISED query processing approaches which parse, normalise and split the query

into sub-queries, determine for each part of the query the relevant sources and eventually evaluate the sub-queries directly against those sources. These systems can be categorised into approaches which perform (i) query federation over endpoints with the SPARQL protocol, (ii) process queries in totally decentralised peer-to-peer (P2P) networks, or (iii) HTTP `GET` lookups to retrieve the necessary source content. Next, we present the related work for each of the categories in Figure 2.3 from left to right, starting with centralised query approaches.

2.3.1 Centralised Approaches

Traditional SPARQL query approaches for Linked Data – the ones directly codified by SPARQL semantics – locally replicate the content of the remote Linked Data sources, e.g., in a triple store and execute the SPARQL queries over the local copy. The primary challenges of a centralised approach are (i) to have as much coverage of the Web of Data as possible, (ii) to keep results up to date, (iii) to be able to process potentially expressive (i.e., expensive) SPARQL queries in an efficient manner and with high concurrency. Such approaches typically feature a crawler or other data acquisition component which, e.g., follows links between documents to discover new information, and/or downloads documents which have been requested for indexing by remote parties. In previous years, we have supported such a service powered by YARS2 [Harth et al., 2007b], which allows for querying millions of RDF Web documents (and their entailments), but have since discontinued this project due to prohibitive running costs for our research hardware. Current centralised stores harvesting Linked Data include OpenLink's LOD cache[8], powered by the Virtuoso quad store [Erling and Mikhailov, 2009], the FactForge [Bishop et al., 2011b] SPARQL store[9] which includes materialised data supported by BigOWLIM [Bishop et al., 2011a], and more recently the Sindice [Oren et al., 2008] SPARQL store[10], again powered by Virtuoso.

A recent system using B^+-trees to index RDF data is RDF-3X [Neumann and Weikum, 2008, 2010]. To answer queries with variables in any position of an RDF triple, RDF-3X holds indexes for querying all possible combinations of subject, predicate and object – an idea introduced in [Harth and Decker, 2005]. RDF-3X uses sophisticated join optimisation techniques based on statistics derived from the data. In contrast to this work, both approaches (and similarly Hexastore [Weiss et al., 2008]) use a different data structure for the index and focus on centralised RDF stores rather than distributed Linked Data sources. That said, a strategy which completely indexes RDF triples and their sources and performs lookups to validate results derived from the (possibly outdated) indexes is possible. However, it involves overhead in creating, storing, and maintaining the complete indexes which we avoid with more lightweight index structures.

The above objectives are (partially) met using distribution techniques, data replication, optimised indexes, compression techniques, data synchronisation, and so on [Bishop et al., 2011a; Erling and Mikhailov, 2009; Harth et al., 2007b; Oren et al., 2008]. Nevertheless, given that such services often index millions of documents, they often require large amounts of resources to run. In particular, maintaining a local, *up-to-date* index with good coverage of the Web of Data is a Sisyphean task.

Centralised approaches provide excellent query response times due to extensive preprocessing carried out during the load and indexing steps, but suffer from the following core drawbacks. First, the aggregated data is never current as the process of collecting and indexing vast amounts of data is time-consuming. Second,

8 `http://lod.openlinksw.com/sparql`
9 `http://factforge.net/sparql`
10 `http://sparql.sindice.com/`

from the viewpoint of a single query, the materialisation involves unnecessary data gathering, processing, and storage since large portions of the data might not be used for answering the particular query. Lastly, due to replicated data storage, data providers have to give up sole sovereignty on their data (e.g., they cannot restrict or log access anymore since queries are answered against a copy of the data).

2.3.2 Decentralised Approaches

The second general category of query approaches for Linked Data assumes that query relevant data is distributed in a decentralised infrastructure, e.g., as the Web of Data, or in Peer-to-peer systems. We identified three sub categories for such decentralised query approaches and discuss in this section two out of the three variations, namely SPARQL query federation and RDF query processing in P2P systems. Our work falls into the remaining category, the lookup based query execution approach, which we will discuss afterwards in a separate section.

2.3.2.1 *SPARQL Federation*

In general, decentralised Linked Data query processing can be considered as a special case of federated query processing [Hose et al., 2011; Ladwig and Tran, 2010]. However, traditionally, federation-based query processing approaches typically assume a relatively small number of autonomous sources (holding the sovereignty of their data) and that processing power is attainable at the sources themselves, which could be leveraged in parallel for query processing. Such distributed or federated approaches [Heimbigner and McLeod, 1985] offer several advantages: the system is more dynamic with up-to-date data and new sources can be added and the systems require less storage and processing resources at the site that issues the query. The potential drawback, however, is that these systems cannot give strict guarantees on query performance since the integration process relies on a large number of potentially unreliable query providers.

The same applies in general for federated Linked Data query processing which considers a scenario in which queries are executed over a distributed set of SPARQL stores: sources with query processing capabilities. We see query federation as related, but not really in the scope of our work since we focus on the query execution over millions of loosely connected, decentralised and small data sources without any capabilities to answer or process queries. Thus, we briefly capture recent development in SPARQL federation.

A primary challenge for federated SPARQL engines is to decide which parts of a query are best routed to which store. Some systems, such as SPARQL DQP [Aranda et al., 2011], require that sub-queries are annotated with the SPARQL 1.1 `SERVICE` keyword, which allows users to invoke remote stores and, more generally, to state which parts should be routed where. Other federated SPARQL engines locally index "service descriptions" or "catalogues", which describe the contents of remote stores and are used to split and route sub-queries without explicit `SERVICE` annotations [Quilitz and Leser, 2008]. One of the earliest works going in this direction (and which predated SPARQL by over three years) was by Stuckenschmidt et al. [2004], who proposed summarising the content of distributed RDF repositories using schema paths (non-empty property chains). The SemWIQ architecture [Langegger et al., 2008] uses counts of the extension of each class and property in a store to create a catalogue used for routing queries; later work extended the set of available statistics using a tool called RDFStats [Langegger and Wöß, 2009], which also provides histograms covering e.g., subjects or data types, as well as estimated cardinalities of selected (sub-)queries. SPLENDID [Görlitz and Staab, 2011] is another

federation infrastructure, which uses the Vocabulary of Interlinked Datasets (voiD) to describe the content of endpoints [Alexander and Hausenblas, 2009]. An alternative to pre-computed service descriptions is proposed by FedX [Schwarte et al., 2011], which instead probes SPARQL endpoints with `ASK` sub-queries to see if they have relevant information during query-time (this information can be cached and re-used).

In recognition of the growing popularity of federated SPARQL, the W3C Working Group has added new federation features [Harris and Seaborne, 2012]. As already mentioned, the `SERVICE` keyword can be used to invoke remote SPARQL endpoints, and the new `VALUES` (previously known as `BINDINGS`) feature can be used to "ship" sets of intermediate bindings to an endpoint [Harris and Seaborne, 2012]. In addition, the "SPARQL 1.1 Service Description" proposes a vocabulary for describing the functionalities and datasets of SPARQL endpoints in a standard way [Williams, 2012].

2.3.2.2 *P2P Query Approach*

There also exists some approaches to query RDF data in a totally distributed and decentralised setup, similar to the Web but unlike federation or lookup based query approaches. Such query processing is a main target in the world of P2P. However, these approaches cannot be deployed for the current Linked Data infrastructure and would require SPARQL processing functionality at the participating nodes, similar to the SPARQL federation scenario. As such, the works in the area of SPARQL query processing over P2P infrastructures are not in the scope of this thesis, however we briefly highlight interesting approaches.

In [Heine, 2006; Heine et al., 2005], the whole RDF model graph is mapped to nodes of a distributed hash table (DHT). DHT lookups are used to locate triples. This implements a rather simplistic query processing strategies based on query graphs that can be mapped to the model graph. Query processing basically conforms to matching query graph and model graph. RDF-Schema data are also indexed in a distributed manner and used by applying RDF-Schema entailment rules. On the downside, query processing has two subsequent phases and sophisticated query constructs that leverage the expressiveness of queries are not supported. There exist several other proposals for large-scale distributed RDF stores [Stuckenschmidt et al., 2005, 2004], RDF querying in Peer-to-peer environments [Nejdl et al., 2002], including approaches supporting partial RDF Schema inference [Adjiman et al., 2006]. RDFPeers [Cai and Frank, 2004] is another distributed infrastructure for managing large-scale sets of RDF data. Similar to, for instance, [Karnstedt et al., 2007], each part of a triple is indexed, but whole triples are stored each time. Numerical data are hashed using a locality-preserving hash function. Load-balancing is discussed as well. Queries formulated in formal query languages, such as RDQL [Miller et al., 2002], can be mapped to the supported native queries. RDFPeers supports only exact-match queries, disjunctive queries for sets of values, range queries on numerical data, and conjunctive queries for a common triple subject. Query resolution is done locally and iteratively. [Karnstedt, 2009; Karnstedt et al., 2007] extends this to sophisticated database-like query processing capabilities, total decentralisation, and considers also data heterogeneity. GridVine [Aberer et al., 2004; Cudré-Mauroux et al., 2007] is a peer data management infrastructure addressing both scalability and semantic heterogeneity. Scalability is addressed by peers organised in a structured overlay network forming the physical layer, in which data, schemata, and schema mappings are stored. Semantic interoperability is achieved through a purely decentralised and self-organising process of pairwise schema mappings and query reformulation. This forms a semantic mediation layer on top and independent of the physical layer. GridVine supports triple pat-

tern queries with conjunction and disjunction, implemented by distributed joins across the network. It does not apply the idea of cost-based database-like query processing over multiple indexes. We see the distribution of query processing load as a promising way to achieve real scalability, data freshness and data authority.

Unstructured P2P systems leverage statistical data for source selection using so-called routing indexes. Crespo et al. [Crespo and Garcia-Molina, 2002] introduced the notion of routing indexes in P2P systems as structures that, given a query, return a list of interesting neighbours (sources) based on a data structure conforming to lists of counts for keyword occurrences in documents. Based on this work, other variants of routing indexes have been proposed, e.g., based on one-dimensional histograms [Petrakis et al., 2004], Bloom Filters [Petrakis and Pitoura, 2004], bit vectors [Marzolla et al., 2006], or the QTree [Hose et al., 2005]. A common feature across these systems is to use a hash function to map string data to a numerical data space. In contrast to our work, the focus of query optimisation in P2P systems is to share load among multiple sites and on local optimisation based on routing indexes.

However, a main problem in the current world of Linked Data sources is that they are usually quite restricted in query processing capabilities. All the decentralised P2P approaches require SPARQL processing functionality at the participating nodes which is not given at the current Web of Data.The mentioned decentralised approaches, however, bear high potential for designing scalable distributed index structures where single sources can connect to kind of super-nodes. These super-nodes form the actual index and are responsible for query routing and processing. An interesting work in this context with promising achievements focusing on Semantic Web technology is presented by Schlosser et al. [2002], which imposes a scalable and self-managing structured overlay on the participating nodes.

2.3.3 Lookup-based Approaches

With the increasing popularity and availability of Linked Data, a new type of decentralised query approach have emerged which exploit the underlying Linked Data principles. These principles can be used to identify and retrieve query relevant sources based on a correspondence between result URIs and source URIs. The targeted data sources do not provide query processing capabilities and interfaces and as such, queries can be only executed over retrieved/dereferenced content by performing lookups on the source URIs.

Such, *lookup-based query approaches* directly access remote data sources at runtime to dynamically select, retrieve and build an integrated dataset over which SPARQL queries can be evaluated. People sometimes consider the lookup-based query evaluation as a query federation scenario. However, strictly speaking, the sources are mostly not aware that they are part of the query architecture and thus, the actual query execution is performed in a mediator approach rather than as federation [Hose et al., 2011].[11]

Ladwig and Tran [2010] identified and named three conceptual approaches for that lookup-based querying which differ in how query relevant sources are selected. The following classification can be distinguished:

i) The *link traversal* based query execution does not require information about the content of sources before the query evaluation. Query relevant sources are discovered and retrieved in an explorative process by *traversing links* in already collected data in a bottom up fashion.

11 This is also true for some of the SPARQL federation approaches.

ii) An *index-based*, top-down query evaluation assumes that source descriptions are available and are used to select query relevant sources before the query evaluation. The query engines retrieved the content of the selected sources and evaluate the query over the integrated data graph.

iii) The third approach is based on a *mixed strategy* of top-down and bottom-up source selection assumes that not all available sources are known before the query execution and new sources can be discovered during the evaluation of the query.

2.3.3.1 *Link traversal based Query Execution*

The first lookup-based query approach we introduce, processes a given query directly over the Web without any source knowledge prior to query execution and discovers query-relevant sources on-the-fly during the evaluation of queries This approach was originally propose by Hartig et al. [2009] and is referred to as the "*Link Traversal Based Query Execution*" (LTBQE) [Hartig, 2011b]. Analysis and extensions of LTBQE forms a core part of this thesis.

The LTBQE technique uses dereferenceable URIs in the query – and recursively, in the intermediate results – to automatically determine a focussed set of sources which, by Linked Data principles, are likely to be *query relevant*, retrieving them for answers.

Given a SPARQL query, the core operation of LTBQE is to identify and retrieve a focused set of query-relevant RDF documents from the Web of Data from which answers can be extracted. The approach begins by dereferencing URIs found in the query itself. The documents that are returned are parsed, and triples matching patterns of the query are processed; the URIs in these triples are also dereferenced to look for further information, and so forth. The process is recursive up to a fix-point wherein no new query-relevant sources are found. New answers for the query can be computed on-the-fly as new sources arrive. When operating over compliant Linked Data, this approach often bypasses the need for source graphs to be explicitly named or pre-indexed, allowing for *ad hoc*, live discovery. Since no local index is required, this approach can be used in decentralised scenarios, where clients can execute queries remotely over the Web without accessing a centralised service. The unique challenges for such an approach are (i) to find as many query-relevant sources as possible to improve recall of answers; (ii) to conversely minimise the amount of sources accessed to avoid traffic and slow query-response times; (iii) to optimise query execution in the absence of typical selectivity estimates, etc. [Hartig, 2011b; Hartig et al., 2009].

The theoretical foundation for the explorative LTBQE approach was published by Bouquet et al. [2010] and further developed by Hartig and Freytag [2011]; a comprehensive analysis about the semantics and computability is provided by Hartig [2012]. We now formally define the key notion of query-relevant documents in the context of LTBQE, and give an indication as to how these documents are derived. This is similar in principle to the generic notion of reachability introduced previously [Hartig, 2012; Hartig and Freytag, 2011], but relies here on concrete HTTP specific operations:

Definition 8 (Query Relevant Sources & Answers)**.** *First let* $\mathsf{uris}(\mu) \coloneqq \{u \in \mathbf{U} \mid \exists v \text{ s.t. } (v, u) \in \mu\}$ *denote the set of URIs in a solution mapping* μ*. Given a query* Q *and an intermediate dataset* Γ*, we define the function* qrel*, which extracts from* Γ *a set of URIs that can (potentially) be dereferenced to find further sources deemed relevant for* Q*:*

$$\mathsf{qrel}(Q, \Gamma) \coloneqq \bigcup_{tp \in Q} \bigcup_{\mu \in [\![\{tp\}]\!]_\Gamma} \mathsf{uris}(\mu)$$

To begin the recursive process of finding query-relevant sources, LTBQE takes URIs in the query – denoted with $\mathcal{U}_Q \coloneqq \mathsf{terms}(Q) \cap \mathbf{U}$ *– as "seeds", and builds an initial dataset by dereferencing these URIs:* $\Gamma_0^Q \coloneqq \mathsf{derefs}(\mathcal{U}_Q)$. *Thereafter, for* $i \in \mathbb{N}$, *define:*[12]

$$\Gamma_{i+1}^Q \coloneqq \mathsf{derefs}\big(\mathsf{qrel}(Q, \Gamma_i^Q)\big) \cup \Gamma_i^Q$$

The set of LTBQE query relevant sources *for* Q *is given as the least* n *such that* $\Gamma_n^Q = \Gamma_{n+1}^Q$, *denoted simply* Γ^Q. *The set of* LTBQE query answers *for* Q *is given as* $[\![Q]\!]_{\Gamma^Q}$, *or simply denoted* $[\![Q]\!]$.

Example 2.17. We illustrate this core concept of LTBQE query-relevant sources with a simple example based on Figure 2.1. Let Q be the following query looking for the names of the authors of a named paper:

```
SELECT ?authorName
WHERE {
  dblpP:HartigBF09 foaf:maker ?author .
  ?author foaf:name ?authorName .
}
```

Query 2.2: Names of paper authors.

First, the process extracts all raw query URIs, resulting in:
$\mathcal{U}_Q = \{$`dblpP:HartigBF09`, `foaf:name`, `foaf:maker` $\}$.
In the next stage, the engine dereferences these URIs. Given that redirs(`dblpP:HartigBF09`) = `dblpPDoc:HartigBF09` & redirs(`foaf:maker`) = redirs(`foaf:made`) = `foafSpec:`, dereferencing $\mathcal{U}_Q$ results in two unique named graphs, *viz.*: $\big($`dblpPDoc:HartigBF09`, get(`dblpPDoc:HartigBF09`)$\big)$ and $\big($`foafSpec:`,get(`foafSpec:`)$\big)$. These two named-graphs comprise Γ_0^Q. (In fact, only the former graph will ultimately contribute answers.)

Second, LTBQE looks to extract additional query relevant URIs by seeing if any query patterns are matched in the current dataset. By reference to the graph `dblpPDoc:HartigBF09` in Figure 2.1, we see that for the pattern "`dblpP:HartigBF09 foaf:maker ?author .`", the variable `?author` is matched by two unique URIs, namely `dblpA:Christian_Bizer` and `dblpA:Olaf_Hartig`, which are added to $\mathsf{qrel}(Q, \Gamma_0^Q)$. Nothing else is matched. Hence, these two URIs are dereferenced and the results added to Γ_0^Q to form Γ_1^Q.

LTBQE repeats the above process until no new sources are found. At the current stage, Γ_1^Q now also contains the two sources `dblpADoc:Christian_Bizer` and `dblpADoc:Olaf_Hartig` needed to return:

?authorName
"Christian Bizer"
"Olaf Hartig"

Furthermore, no other query-relevant URIs are found and so a fix-point is reached and the process terminates: $[\![Q]\!]$ contains the above results.

The LTBQE approach has an inherent trade-off between the number of sources accessed and the recall of the response (the percentage of globally available answers returned), which varies from accessing no sources and returning no results,

12 In practice, URIs need only be dereferenced once; *i.e.*, only URIs in $\mathsf{qrel}(Q, \Gamma_i^Q) \setminus (\mathsf{qrel}(Q, \Gamma_{i-1}^Q) \cup \mathcal{U}_Q)$ need be dereferenced at each stage.

to (theoretically) processing the entire Web of Data. Furthermore, LTBQE relies on Linked Data principles as cues to identify a minimal amount of sources that maximise results: however, Linked Data principles are not always fully adhered to on the Web.

2.3.3.2 *Index/Catalog-based Query Processing*

The index-based, top-down evaluation determines all query relevant sources prior-to the actual query execution, falling back upon local knowledge about the content of sources. The index holds (meta)information and statistics about remote Web sources and is referred to as a "*source-selection index*". These specialised indexes may be implemented as a simple inverted-index structure [Li and Heflin, 2010; Oren et al., 2008], a query-routing index [Tran et al., 2010] or a schema-level index [Stuckenschmidt et al., 2004]. We will compare these index structures as to how well suited they are for the source selection task in Chapter 5. Furthermore, we exploit hash-based index structures that summarises the data of or statistics about source in a very compact form which then can be exploited to optimise queries or for selecting query relevant sources. These hash-based index structures are inspired by works on data summaries in the database community.

DATA SUMMARIES Database systems have exploited the idea of capturing statistics about data for many years by using histograms [Ioannidis, 2003], primarily for selectivity and cardinality estimates over local data. The first histograms were only defined on one attribute value. In reality, one usually observes correlation between attributes and the assumption of independence often leads to bad approximations of result cardinalities [Poosala and Ioannidis, 1997]. The first histogram developed to counteract this problem was the two-dimensional equi-depth histogram proposed in [Muralikrishna and DeWitt, 1988]. First approaches for incremental maintenance of one-dimensional equi-depth and compressed histograms were proposed in [Gibbons et al., 2002]. Controversially, most multidimensional histograms [Gunopulos et al., 2000; Muralikrishna and DeWitt, 1988; Poosala and Ioannidis, 1997] are static and need to be reconstructed each time the data they summarise is updated. Some of such histograms [Bruno et al., 2001; Srivastava et al., 2006] even allow overlapping buckets, which are adapted during runtime. However, these approaches use one base bucket covering the whole data space to represent all the data that is not represented by separate buckets. We investigate use of multi-dimensional histograms for index-based lookup query processing over Linked Data in Chapter 5.

There exist several other summary techniques for large-scale multidimensional data that could be applied in this setup. There is a wide range of other hash-based structures, such as extendible hashing and linear hashing [Huang, 1985; Rathi et al., 1990]. Other interesting summary techniques that were mainly designed for the data-stream scenario, such as wavelets and sketches [Babcock et al., 2002]. In data streams, these techniques that mainly used for approximate query answering, particularly for aggregation queries. Sketches represent a summary of a data stream using a very small amount of memory. They are typically used to answer distance queries, but also to estimate the number of unique values, e.g., for the estimation of the size of a self-join [Babcock et al., 2002]. Gilbert et al. [2001] shows how to use sketches to compute wavelet coefficients efficiently. However, there exist other methods to compute wavelets as summaries for data streams in a single pass. [Chakrabarti et al., 2001] shows how wavelets can be used for selectivity estimation and that they are usually more accurate than histograms. The authors show how to achieve approximate query processing by computing joins, aggregations and selections entirely on the wavelet coefficients. A main advantage of wavelets

and sketches is that they are designed to support particularly efficient construction and maintenance phases. In contrast, a main advantage of the multidimensional structures discussed in this thesis is that they inherently capture data dependencies and were initially designed for multidimensional indexing, rather than approximate query answering on streams.

2.3.3.3 *Mixed*

The third strategy involves a combination of top-down and bottom-up techniques. This strategy uses (in a top-down fashion) knowledge about sources to map query terms or query sub-goals to sources which can contribute answers, then discovering additional query relevant sources using a bottom-up approach. The approach of Ladwig and Tran [Ladwig and Tran, 2010] exploits different types of knowledge of sources available beforehand, and also, incorporates information gained during query processing. The results show that the mixed approach efficiently reports results earlier than the explorative bottom-up approach and also reduces the query time. The same authors optimised this approach with an efficient join operator based on symmetric hash joins [Ladwig and Tran, 2011] which resulted in a reported speed up of the query time by up to 70%. In relation to our work, later presented in Chapter 6, these mixed approaches focussed solely on optimising the query performance compare the our proposed hybrid framework which investigate and explore the mixed strategy from a different angle. To the best of our knowledge, our contribution in Chapter 6 is the first which studies how the mixed strategy can improve not only the query time but also the freshness and recall of results.

3 ON THE DYNAMICS OF LINKED DATA

"You should have started this experiment two years ago!"

— Aidan Hogan, *2012*

In this chapter, we introduce the evaluation dataset(s) of this thesis and present two experiments which study the dynamicity of Linked Data and how the ever-changing Web prevents materialisation-based query engines from guaranteeing up-to-date results.

In order to design our experiments, we first ask the important question: "What is the Web of Data?" in Section 3.1. We compare two prominent "views" thereof; (1) the Billion Triple Challenge dataset view, and (2) the CKAN/LOD cloud view. The findings of this study provide us with an evaluation dataset used throughout the thesis to benchmark the proposed approaches in Chapter 4 – Chapter 6.

Next, we present our contribution to the young research area of Linked Data dynamics. In Section 3.2, we verify the assumption that Linked Data is dynamic and that we can find different levels of dynamicity for different sources. In Section 3.3, we measure how consistent the cached data of two prominent public SPARQL store are compared to the data on the Web and confirm the problem of materialised indexes serving stale results. The results of our second experiment are further used in Chapter 6 as input for the query planner of our hybrid query framework.

3.1 WHAT IS THE WEB OF DATA?

The Web of Data consists of various data providers which publish varying amounts of information as RDF on the Web, preferably agreeing to the Linked Data principles. However, we observe that some data providers are compliant with Linked Data principles to varying degrees [Hogan et al., 2012a]. Thus, there's no one yardstick by which a dataset can be unambiguously labelled as "Linked Data".

For the purposes of the seminal Linked Open Data (LOD) project, Bizer et al. [2011] use a variety of minimal requirements a dataset should meet in order to be included in the LOD Cloud diagram [1], which serves as a high level overview of connections between Linked Data corpora. However, the LOD Cloud is biased towards large monolithic datasets published on one domain, and does not cover low-volume cross-domain publishing as common for Web vocabularies such as FOAF, SIOC, etc. For example, social micro-blogging platforms like `identi.ca` or `status.net`, content management systems such as Drupal, blog engines like Wordpress, etc., can export compliant, decentralised Linked Data – using vocabularies such as FOAF and SIOC – from the various domains where they are deployed, but their exports are not in the LOD Cloud.

A broader notion to consider is the *Web of Data*, which would cover these latter exporters and vocabularies, but which is somewhat ambiguous and with ill-defined borders. For the main purposes of this chapter in assessing the dynamicity of Linked Data, we define the Web of Data as being comprised of interlinked RDF data published on the Web.[2] No clear road-map is available for the Web of Data per

1 `http://richard.cyganiak.de/2007/10/lod/`

2 This definition is perhaps more restrictive than some interpretations where, e.g., Sindice incorporates Microformats into their Web of Data index [Delbru, 2009].

this definition; the LOD cloud only covers prominent subsets thereof. Perhaps the clearest picture about Linked Data on the Web comes from crawlers that harvest RDF from the Web by traversing links. A prominent example are the Billion Triple Challenge datasets [Bizer and Maynard, 2011], which are made available every year and comprises of data collected during a deep crawl of RDF/XML documents on the Web of Data. However, the precise composition of such datasets is unclear, and requires further study.

To get a better insight, we contrast two such perspectives of the Web of Data. In detail, we compare the:

BILLION TRIPLE CHALLENGE 2011 DATASET (BTC)

> The Billion Triple Challenge 2011 dataset is collected from a Web crawl of over seven million RDF/XML documents in 2011 [Harth and Maynard, 2012];

COMPREHENSIVE KNOWLEDGE ARCHIVE NETWORK (CKAN)

> The Comprehensive Knowledge Archive Network is a repository – specifically the `lodcloud` group therein – containing high-level metrics reported by Linked Data publishers, used in the creation of the LOD Cloud [Pollock et al., 2004].

We want to see how these two different views of the Web of Data compare and how they can be used as the basis for an evaluation corpus.

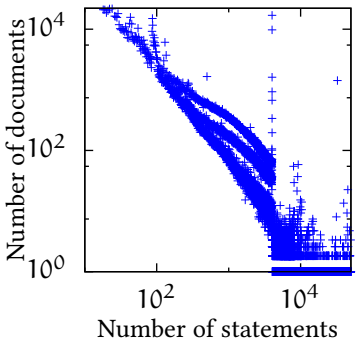
3.1.1 The BTC 2011 Dataset

The BTC dataset is crawled from the Web of Data, previously using our Multi-Crawler framework [Harth et al., 2006] and recently the LDSpider [Isele et al., 2010] framework, for the annual Billion Triple Challenge [Bizer and Maynard, 2011] at the International Semantic Web Conference (ISWC). The dataset empirically captures a deep, broad sample of the Web of Data in situ.

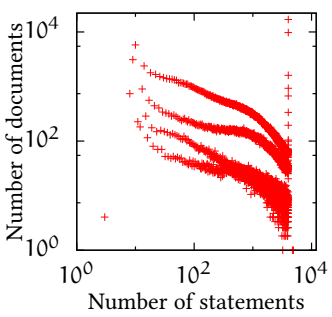
However, the details of how the Billion Triple Challenge dataset is collected are somewhat opaque. The seed list is sampled from the previous year's dataset [Harth and Maynard, 2012], where one of the initial seed-lists in past years was gathered from various semantic search engines. The crawl is for RDF/XML content, and follows URIs extracted from all triple positions. Scheduling (i.e., prioritising URIs to crawl) is random, where URIs are shuffled at the end of each round. As such, any RDF/XML document reachable through other RDF/XML documents from the seed list is within scope; otherwise, what content is (or is not) in the BTC – and how "representative" the dataset is of the Web of Data – is difficult to ascertain purely from the collection mechanisms.

As such, it is more pertinent to look at what the dataset actually contains. The most recent BTC dataset available during the time of writing was crawled in May and June 2011. The final dataset contains 2.145 billion quadruples, extracted from 7.411 million RDF/XML documents. The dataset contains RDF documents sourced from 791 *pay-level domains* (cf. Definition 1). The BTC 2011 dataset contained documents from 240,845 fully qualified domain names (cf. Definition 1), 233,553 of which were from the `livejournal.com` PLD.

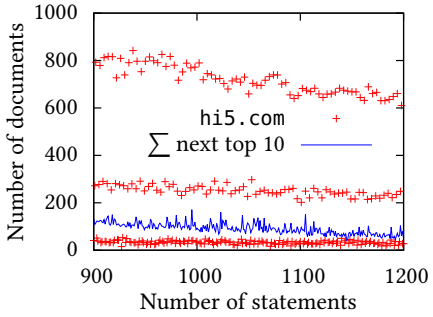
On the left-hand side of Table 3.1 we enumerate the top-25 PLDs in terms of quadruples contributed to the BTC 2011 dataset. Notably, a large chunk of the dataset (~64 %) is provided by the `hi5.com` domain: a social gaming site that exports a FOAF file for each user. As observed for similar corpora (cf. [Hogan et al., 2011, Table A.1]) `hi5.com` has many documents, each with an average of over two thousand statements – an order of magnitude higher than most other domains – leading it to dominate the overall volume of BTC statements.



(a) Overall



(b) `hi5.com`



(c) `hi5.com` dominates

Figure 3.1: Distribution of the number of statements in documents for the BTC2011 dataset (3.1a) and (3.1b) for `hi5.com`; as well as (3.1c) the periodicity of distribution of statements per document for `hi5.com` that causes the split tail in (3.1a) & (3.1b).

The dominance of the data provider `hi5.com` – and to a lesser extent similar sites like `livejournal.com` – shape the overall characteristics of the BTC 2011 dataset. To illustrate one prominent such example, Figure 3.1a gives the distribution of statements per document in the BTC dataset on log/log scale, where one can observe a rough power-law(-esque) characteristic. However, there is an evident three-way split in the tail emerging at about 120 statements, and ending in an outlier spike at around 4,000 statements. By isolating the distribution of statements-per-document for `hi5.com` in Figure 3.1b, we see that it contributes to the large discrepancies in that interval. The stripes are caused by periodic patterns in the data, due to its uniform creation: on the `hi5.com` domain, RDF documents with a statement count of $10 + 4f$ are heavily favoured, where ten triples form the base of a user's description and four triples are assigned to each of f friends. Other lines are formed due to two optional fields (`foaf:surname`/`foaf:birthday`) in the user profile, giving a $9 + 4f$ and $8 + 4f$ periodicity line. An enforced ceiling of $f \leqslant 1,000$ friends explains the spike at (and around) 4,010.

The core message is that although the BTC offers a broad view of the Web of Data, covering 791 domains, in absolute statement-count terms, the dataset is skewed by a few high-volume exporters of FOAF, and in particular `hi5.com`. When deriving global statistics and views from the BTC, the results say more about the code used to generate `hi5.com` profiles than the efforts of thousands of publishers.[3] This is also a naturally-occurring phenomenon in other corpora (e.g., [Ding and

3 Furthermore, `hi5.com` is not even a prominent domain on the Web of Data in terms of being linked, and was ranked 179/778 domains in a PageRank analysis of a similar corpus; `http://aidanhogan.com/ldstudy/table21.html`

№	Top-25 BTC			Top-25 CKAN/LOD		
	PLD	*BTC*	*LOD*	*PLD*	*LOD*	*BTC*
1	hi5.com	1,371,854,358	–	rpi.edu	9,803,140,000	900,464
2	livejournal.com	169,863,721	–	linkedgeodata.org	3,000,000,000	–
3	tfri.gov.tw	153,300,321	23,015,257	legislation.gov.uk	1,900,000,000	31,990,934
4	scinets.org	56,075,080	–	wright.edu	1,730,284,735	5
5	ontologycentral.com	55,124,003	122,000,000	concordia.ca	1,500,000,000	–
6	rdfize.com	36,154,381	–	data.gov.uk	1,336,594,576	13,302,277
7	legislation.gov.uk	31,990,934	1,900,000,000	dbpedia.org	1,204,000,000	25,776,027
8	identi.ca	30,429,795	–	rdfabout.com	1,017,648,918	–
9	bibsonomy.org	28,670,581	–	dbtune.org	888,089,845	1,634,891
10	dbpedia.org	25,776,027	1,204,000,000	uniprot.org	786,342,579	4,004,440
11	freebase.com	25,488,720	337,203,427	unime.it	586,000,000	–
12	opera.com	23,994,423	–	uriburner.com	486,089,121	–
13	bio2rdf.org	20,168,230	72,585,132	openlibrary.org	400,000,000	25,396
14	archiplanet.org	13,394,199	–	sudoc.fr	350,000,000	–
15	data.gov.uk	13,302,277	1,336,594,576	freebase.com	337,203,427	25,488,720
16	loc.gov	7,176,812	24,151,586	fu-berlin.de	247,527,498	5,658,444
17	vu.nl	6,106,366	14,948,788	dataincubator.org	205,880,247	3,695,950
18	bbc.co.uk	5,984,102	80,023,861	viaf.org	200,000,000	–
19	rambler.ru	5,773,293	–	europeana.eu	185,000,000	–
20	fu-berlin.de	5,658,444	247,527,498	moreways.net	160,000,000	–
21	uniprot.org	4,004,440	786,342,579	rkbexplorer.com	134,543,526	220
22	dataincubator.org	3,695,950	205,880,247	ontologycentral.com	122,000,000	55,124,003
23	zitgist.com	3,446,077	60,000,000	opencorporates.com	100,000,000	–
24	daml.org	3,135,225	–	uberblic.org	100,000,000	–
25	mybloglog.com	2,952,925	–	geonames.org	93,896,732	458,490

Table 3.1: Statement counts for top-25 PLDs in the BTC with corresponding reported triple count in CKAN (left), and top-25 PLDs in CKAN with BTC quad count (right)

Finin, 2006; Hogan et al., 2011]) crawled from the Web of Data – not just isolated to the BTC dataset(s) – and is *not* easily fixed. One option to derive meaningful statistics about the Web of Data from such datasets is to apply (aggregated) statistics over individual domains, and never over the corpus as a whole.

3.1.2 CKAN/LOD Cloud Metadata

In contrast to the crawled view of the Web of Data, the CKAN repository indexes publisher-reported statistics about their dataset. These CKAN metadata are then used to decide eligibility for entry into the LOD cloud [Bizer et al., 2011]: a highly prominent depiction of Linked Open Datasets and their interlinkage. A CKAN-reported dataset is listed in the LOD cloud iff it fulfils the following requirements: the dataset has to (1) be published according to core Linked Data principles, (2) contain at least one thousand statements and (3) provide at least 50 links to other LOD cloud datasets[4].

Given the shortcomings of the crawled perspective on the Web of Data, we explore these self-reported metadata to acquire an alternative view. On September 29, 2011, we downloaded the meta-information for the datasets listed in the `lodcloud` group on CKAN[5]. The data contain example URIs for the dataset and statistics such as the number of statements. We discovered dataset description for 297 datasets, spanning 206 FQDNs and 133 PLDs. On the right hand side of Table 3.1, we enumerate the top-25 largest reported datasets in the `lodcloud` group on CKAN. Note that where multiple datasets are defined on the same domain, the triple count is presented as the summation of said datasets. In this Table, we see a variety of domains claiming to host between 9.8 billion and 94 million triples.

4 `http://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/DataSets/CKANmetainformation`

5 `http://thedatahub.org/group/lodcloud`

PLD	Robots	Http-401	Http-502	Mime	Unreachable	Other
linkedgeodata.org			X	X		
concordia.ca				X		
rdfabout.com				X		
unime.it					X	
uriburner.com	X					
sudoc.fr					X	
viaf.org				X		
europeana.eu						X
moreways.net					X	
uberblic.org		X				

Table 3.2: Reasons for largest ten PLDs in CKAN/LOD not appearing in BTC 2011.

Regarding the data formats present in the LOD cloud, most of the datasets claim to serve RDF/XML data (85 %), 4 % claim to serve RDFa (of which 50 % did not *also* offer RDF/XML) which shows the popularity of RDF/XML. However, the syntax metadata are somewhat unreliable, where improper mime-types are often reported.

3.1.3 Comparison Between BTC and CKAN/LOD

Finally, we contrast the two different perspectives of the Web of Data. Between both, there are 854 PLDs mentioned, with BTC covering 791 domains (~92.6 %), CKAN/LOD covering 133 domains (~15.6 %), and the intersection of both covering 70 domains (~8.2 % overall; ~8.8 % of BTC; ~52.6 % of CKAN/LOD). CKAN/LOD reports a total of 28.4 billion triples, whereas the BTC (an incomplete crawl) accounts for 2.1 billion quadruples (~7.4 %). However, only 384.3 million quadruples in the BTC dataset (~17.9 %) come from PLDs mentioned in the extracted CKAN/LOD metadata.

In Table 3.1, we present the BTC and CKAN/LOD statement counts side-by-side. We can observe that a large number of high-volume BTC domains are not mentioned on CKAN/LOD, where the datasets in question may not publish enough RDF data to be eligible by CKAN/LOD, or may not follow Linked Data principles or have enough external links, or may not have self-reported. Perhaps more surprisingly however, we note major discrepancies in terms of the catchment of BTC statements versus CKAN/LOD metadata. Given that BTC can only *sample* larger domains, a lower statement count is to be expected in many cases: however, some of the largest CKAN/LOD domains do not appear at all. Reasons can be found through analysis of the BTC 2011's publicly available access log. In Table 3.2, we present reasons for the top-10 highest-volume CKAN/LOD data providers not appearing in the BTC 2011 dataset (i.e., providers appearing with "—" on the right-hand side of Table 3.1). Robots indicates that crawling was prohibited by `robots.txt` exclusions; Http-401 and Http-502 indicate the result of lookups for URIs on that domain; Mime indicates that the content on the domain did not return `application/rdf+xml` used as a heuristic in the BTC crawl to filter non-RDF/XML content; Unreachable indicates that no lookups were attempted on URIs from that domain; Other refers solely to `europeana.eu`, which redirected all requests to their home page.

		BTC
PROS:	✓	covers more domains (791)
	✓	empirically validated
	✓	includes vocabularies
	✓	includes decentralised datasets
CONS:	X	influence of high-volume domains
	X	misses 47.4 % of LOD/CKAN domains
		LOD/CKAN
PROS:	✓	domains pass "quality control"
	✓	community validated
CONS:	X	covers fewer domains (133)
	X	self-reported statistics
	X	misses vocabularies
	X	misses decentralised datasets

Table 3.3: Advantages and disadvantages for both perspectives of the Web of Data.

In summary, we see two quite divergent perspectives on the Web of Data, given by the BTC 2011 dataset and the CKAN/LOD metadata. As enumerated in Table 3.3, both perspectives have inherent strengths and weaknesses. We will use the BTC data sets for the evaluation of our query approaches in Chapter 4 and Chapter 6 because of its broad coverage and diversity and since it includes also vocabularies. However, to measure the dynamicity of Linked Data, we decided to combine both data sets to get the best possible coverage, but on a smaller scale.

3.2 STUDYING THE DYNAMICITY OF LINKED DATA

In this section, we present out experiment to investigate the dynamicity on the Web of Data. Our experiment is inspired by decades of research in the area of dynamicity of the traditional Web, which (mostly) centres around dynamicity on the level of source changes, e.g., Brewington and Cybenko [2000a,b], Lim et al. [2001], Cho and Garcia-Molina [2003a,b], Fetterly et al. [2004] and Ntoulas et al. [2004], amongst others. We refer interested readers to the excellent survey by Oita and Senellart [2011], which provides a comprehensive overview of existing methodologies to detect Web page changes, and also surveys general studies about Web dynamics.

Regarding the study of the dynamics on Linked Data some research regarding dynamics has been conducted with respect to analysing the evolution of ontologies in the life science community [Hartung et al., 2008]. In [Popitsch and Haslhofer, 2011] the authors reported on their work concerning *DSNotify*, a system for detecting and fixing broken links in Linked Data datasets but not considering the patterns or change processes. However – and to the best of our knowledge – we are not aware of any published studies more generally regarding the change frequency of resources on the Linked Open Data Web, and thus deem the experiments herein to be novel.

As such, we design an experiment and aim to provide answers to the following questions about the evolving Web of Data:

CHANGE PROCESS

Q1: *How fast does a source change?*

The study of the change frequency of sources does not only help to get a general understanding about the dynamics but also is very relevant for a broad range of application domains as described already [Cho and Garcia-Molina, 2000a]. Thus, we investigate how long it takes for a source to change.

Q2: *Can the change process be described with a mathematical model?*

Various published studies reported that the *change behaviour* of Web pages corresponds closely with – and can be predicted using – a Poisson distribution [Brewington and Cybenko, 2000a,b; Cho and Garcia-Molina, 2003a,b]. We try to apply the same model to the results from the previous question.

CHANGE CHARACTERISTICS

Q3: *Can we observe different types of changes?*

Given the diversity of publisher domains on the Web, we expect to observe different types of changes, such as updates of sensor readings, only appending information like adding new publications or also a combination of updates, additions and deletions.

Q4: *Can we observe different characteristics for different data providers?*Again, various authors also showed that change frequencies on the traditional Web vary widely across top-level domains [Brewington and Cybenko, 2000a,b; Cho and Garcia-Molina, 2003b; Fetterly et al., 2004]. We also expect that certain Linked Data data providers publish rather static data (e.g., archives or data from statistical institutes) and others rather publish data that change with varying degrees (be it occasional updates in personal files, infrequent bulk updates like in DBPedia, or frequently changing sensor data or social network activities).

Answering these questions requires a large and diverse data set which is constantly monitored over a long timespan to conclude meaningful findings. Despite the fact that we conducted a similar experiment in a previous study [Umbrich et al., 2010a], we are not aware of any significant, heterogeneous and publicly available dataset of Linked Data sources which includes a complete and consistent history of changes over a long timespan. The data corpus for the mentioned study was comparable smaller and, even more important, not diverse enough to obtain a more representative picture. Thus, we (re)designed our own experiment, where we outline our methodologies next.

3.2.1 Methodology

Various aspects must be considered in order to achieve a comprehensive overview of how Linked Data changes and evolves on the Web. Important factors affecting the findings of such an experiment are the chosen dataset, the monitoring frequency and what is actually considered as a data change.

The methodologies used in our evaluation are inspired by legacy related work for Web documents and is based on the experience of our previous study [Umbrich et al., 2010a]. Clearly the design of such an experiment, and gathering the requirements for the resulting collection, is non-trivial: many factors and actors have to be taking into consideration in the context of the open Web and the Linked Data community. Conceptually, we want our experiment to be:

general-purpose:	suitable to study for a wide range of interested parties;
broad:	capturing a wide selection of Linked Data domains;
substantial:	the number of documents monitored should allow for deriving confident statistical measures;
granular & frequent:	offering detailed data on sources;
contiguous:	allowing comparison of sources over time; and
adaptive:	able to discover the arrival of new sources, and can monitor more dynamic sources more frequently.

3.2.1.1 *Sampling a Monitoring Dataset*

Due to the size of the Web and the need for frequent snapshots, sampling is necessary to create an appropriate collection of URIs that can be processed and monitored under the given time, hardware and bandwidth constraints. The goal of our sampling method is thus two-fold: to select a set of URIs that (i) capture a wide cross-section of domains and (ii) can be monitored in a reasonable time given our resources and in a polite fashion. The change frequency of data on the Web can vary significantly across datasets, from rather static sources, such as archives, to high-frequently changing sources, for example in the micro-blogging domain or from sensors. Also, the change volume can range from small-scale updates, in our case, updates involving a low number of triples, to bulk updates, which potentially affect many resources. As such, we wish to combine the BTC/crawled and CKAN/metadata perspectives when defining our seed-list to get a good coverage of data providers and their sources.

We decided to sample our dataset based on a combination of three methods to select URIs from crawled documents: (i) either randomly, (ii) because of specific characteristics (e.g., dynamic or highly ranked), or (iii) to ensure an even spread across different hosts. We start with an initial seed list of 440 URIs taken from: (1) the 220 registered example URIs for the datasets in the LOD cloud at the time of access and complement them with (2) the top-220 document URIs in the BTC dataset of 2011.

The most popular URIs are selected based on a PageRank analysis of the documents in the BTC 2011 dataset, where we select the top-k ranked documents from this analysis (please see [Glimm et al., 2012] for details); note that many of the top ranked documents refer to commonly instantiated vocabularies on the Web of Data, which are amongst the most linked/central Linked Data documents. The resulting core URIs contain 137 PLDs, 120 from the CKAN/LOD examples and 37 from the most popular BTC URIs. This selection guarantees us to cover all relevant domains (similar to [Fetterly et al., 2004]) and to also consider the most popular and interlinked URIs on the Web of Data (similar to [Cho and Garcia-Molina, 2003b]).

Obviously, 440 seed URIs are insufficient to resolve a meaningful corpus for observation over time. Thus, we decide to use a crawl and expand outwards from these core URIs to find other documents to monitor in their vicinity. Importantly, we wish to stay in the close locale of the 440 core URIs; if we go further, we will encounter the same problems as observed for the BTC 2011 dataset, where the data are skewed by a few high-volume exporters. To avoid being diluted by, e.g., `hi5.com` data and the likes, we thus stay within a 2-hop crawl radius from the core URIs.

Starting from our list of 440 core URIs, we expand a 2-hop crawl from which we build the final seed list of URIs to monitor. However, due to unpredictability/non-determinism of remote connections, we want to ensure a maximal coverage of the documents in this neighbourhood. We repeated ten complete 2-hop crawls from

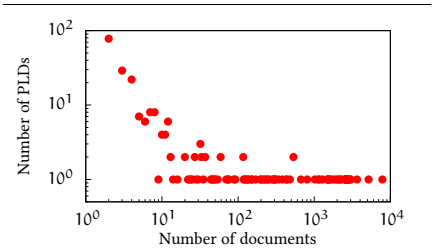


Figure 3.2: Distribution of the number of documents per PLD in the seed list.

№	PLD	URIs
1	gesis.org	7,850
2	chem2bio2rdf.org	5,180
3	dbpedia.org	3,643
4	freebase.com	3,026
5	fer.hr	2,902
6	loc.gov	2,784
7	concordia.ca	2,784
8	dbtune.org	2,767
9	fu-berlin.de	2,689
10	semantictweet.com	2,681

Table 3.4: Top 10 PLDs based on the number of URIs in the seed list.

our core URI list and to ensure comprehensive coverage of URIs in the 2-hop neighbourhood, we take the union of URIs that dereferenced to RDF content, resulting in a total set of 95,737 URIs spanning 652 domains, giving an average of 146.8 dereferenceable URIs per domain.

The result is then our best-effort compromise to achieve representative snapshots of Linked Data that (i) take into account both views on Linked Data by including CKAN and BTC URIs in the core seed list, (ii) extend beyond the core seed list in a defined manner (2 hops), and (iii) do not exceed our crawling resources. Figure 3.2 shows in log/log scale the distribution of the number of PLDs (y-axis) against the number of URIs in the union list (x-axis); we see that 379 PLDs (~58.1 %) have one URI in the list, 78 PLDs (~12.0 %) have two URIs, and so forth. In addition, Figure 3.4 lists the number of URIs for the top-10 PLDs in the set (represented by the ten rightmost dots in Figure 3.2). Technical details about the crawl setup can be found in Käfer et al. [2012].

3.2.1.2 *Monitoring configuration*

Since the Web is undoubtedly evolving in real-time [Brewington and Cybenko, 2000a; Ntoulas et al., 2004], ideally, we would like to monitor the source changes when they happen to get the most accurate information. However, due to the size of the Web and by adhering to server politeness guidelines (posing too many requests to the same domain in the same time frame), it is impossible to monitor all Web sources and react to changes when they happen [Cho and Garcia-Molina, 2000b].

Thus, we model the evolving Web as a *evolving graph process* over a set of *discrete timesteps* $\mathcal{T}$ ($\mathcal{T} = \{t_1, ..., t_n\}, t_i \in \mathbb{N}$). At each timestamp t, we consider that the Web of Data consists of a set of sources and their content. We believe, that this discrete model serves as a good approximation for real-time change processes, depending of course on the granularity of timestamps intervals.

Given our evolving Web model, we now discuss the *monitoring techniques* and intervals we apply.

In general, there exist two fundamental monitoring techniques. The first technique is to periodically download the content of a fixed list of URIs, as widely used in the literature [Brewington and Cybenko, 2000a,b; Cho and Garcia-Molina, 2003a,b]; this technique allows us to study the evolution of sources over time in a *contiguous* fashion. The second technique is to periodically perform a crawl from a defined set of URIs [Ntoulas et al., 2004]; this technique is more suitable if one

wants to study the evolution of the neighbourhood network of the seed URIs in an *adaptive* fashion, but also can introduce a factor of randomness based on the crawling methods.

Again, we decided to apply a hybrid approach: primarily, we do not want to limit our observations to URIs online at the start of the experiment, although they will still be our focus. We thus take the set of 95,737 sampling URIs extracted in the previous section as a *kernel* of *contiguous* URIs accessed consistently in each snapshot. From the kernel, we propose to crawl as many URIs again using the crawler configuration outlined in the previous section, forming the *adaptive* segment of the snapshot. Roughly half of our snapshot would comprise of the contiguous kernel, reliably providing data about said URIs; and the other half of our snapshot would comprise of the adaptive crawl, reflecting changes in the neighbourhood of the kernel. We do not limit PLDs in the adaptive crawl so as to not exclude data providers that come online during the course of the experiment. This setup allows for studying (i) dynamics within the datasets (ii) dynamics between datasets (esp. links) (iii) and the growth of Linked Data and the arrival of new sources (although to a lesser extent).

WEEKLY MONITORING INTERVAL Next, we must decide on the *monitoring intervals* for our platform: how frequently we wish to perform our crawl. In the literature, it is common to take the data snapshots in either a daily [Fetterly et al., 2004; Ntoulas et al., 2004] or weekly [Brewington and Cybenko, 2000a,b; Cho and Garcia-Molina, 2003a,b] fashion. Again, in a practical sense, the intervals are highly dependent on the available resources, and the size of the seed list. Given our resources and the monitoring requirements, we decided to perform a full snapshot every week. We believe this interval to be well within the update latency of current materialised Linked Data query engines.

3.2.1.3 *Change detection function*

A straightforward approach to study the changes on the Web of Data is to compare the sources between two timestamps for which the content is available. However, since Linked Data is used to describe entities and the relation between these, one might be also interested to study how the information about entities change over time as we highlighted in an earlier experiment [Umbrich et al., 2010a]. For this particular experiment, we will ignore entity-centric changes and apply the straightforward source centric change detection.

The change detection of a source between two snapshots t, t' is a trivial task as long as the statements of the resource do not contain blank nodes [Tummarello et al., 2007]. For our evaluation, we used a simple change detection algorithm – based on a merge-sort scan over the weekly snapshots – as follows:

1. skolemise blank nodes within a document, that is, we replace the blank nodes with unique constants based on a deterministic combination of source URL and the blank node itself. In such way, we guarantee that there exists no two blank nodes with the same identifier within the same snapshot;
2. sort all relevant statements for the change detection of an document or entity by their syntactic natural order (subject-predicate-object-[context]);
3. perform pairwise comparison of the statements by scanning two snapshots in linear time;
4. trigger a detection of change as soon as the order of the statements differs between two snapshots (e.g. new statements were added or removed).

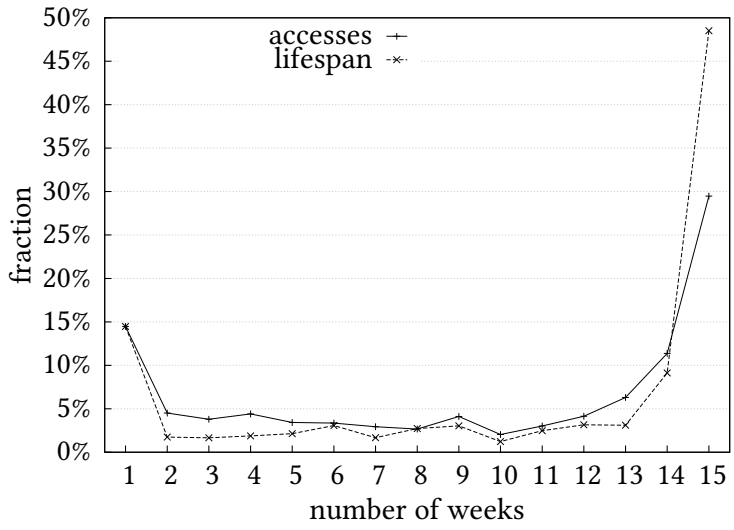


Figure 3.3: Access and lifespan distribution of sources

This algorithms also allows us to categorise the types of changes based on the statements that were added or removed between two snapshots.

3.2.2 Evaluation

In this section, we provide the answers to our four questions regarding the change frequency of resources on the Web of Data.

DATA CORPUS We started our experiment at the beginning of May 2012 and collected so far 15 weekly snapshots at the point of writing. The union of the datasets contains 240,324 distinct sources stemming from 1,351 PLDs.

AVAILABILITY AND LIFESPAN OF SOURCES We give an overview about how many sources are accessed in each monitoring snapshot. In particular, we are interested in characterising the distribution of the number of accesses (i.e., appearances) and the lifespan (i.e., the time interval between the first and last appearance) of the sources in terms of weeks. This computation of the lifespan of a source is slightly different from Cho and Garcia-Molina [2000a], who estimated the lifespan of a document by doubling the time the document was seen in the monitoring window if the document occurred at the beginning of the experiment but not at the end.

Figure 3.3 contains the separate plots of the accesses and lifespan distribution for sources. We can see from the figure that in around 50% of the sources have a lifespan and number of continues accesses of more than 14 weeks. However, the figures do not show how access and lifespan are related.

The figure is not showing the fraction of sources with a certain lifespan and access number. Thus, Table 3.5 shows these details and lists the fraction of sources with their specific lifespan and number of accesses; we omitted the table cells if the fraction of documents was less than 5%. More precisely, ~116k (48.51%) out of the 240k sources have a lifespan of 15 weeks with a total of 70k (29.48%) sources also appearing in each of the 15 weekly snapshots. Still, we found a total of 33.5k (14.47%) documents which we accessed only once during the total time of the ex-

		Lifespan [weeks]				
	№	1	…	14	15	TOTAL
Accesses [weeks]	1	14.47%	…	–	–	14.47%
	⋮					⋮
	13	–	…	1.68%	4.06%	6.3%
	14	–	…	1.78%	9.59%	11.37%
	15	–	…	–	29.48%	29.48%
	TOTAL	14.47%	…	9.13%	48.51%	100 %

Table 3.5: Fraction of documents with specific accesses and lifespan (see Appendix Table B.1 for full results).

№	PLD	URIs
1	identi.ca	3,545
2	freebase.com	3,509
3	dbpedia.org	3,315
4	europa.eu	3,217
5	fer.hr	2,958
6	gesis.org	2,935
7	mcu.es	2,925
8	loc.gov	2,854
9	bbc.co.uk	2,792
10	tfri.gov.tw	2,670

Table 3.6: Top 10 PLDs based on the number of URIs in our evaluation sample.

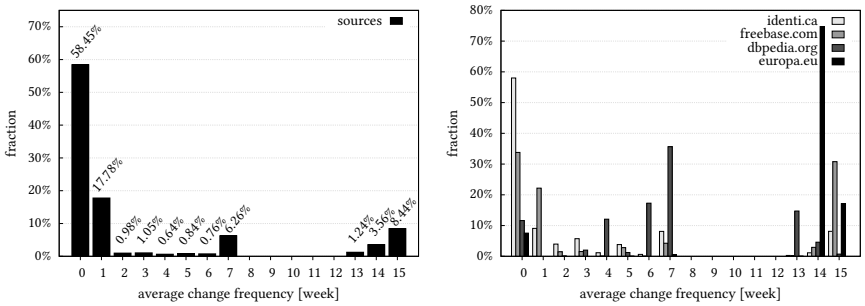
periment. The results for the omitted values are listed in the Appendix in Table B.1 for sources.

For the remainder of our study, we only consider the subset of the corpus which features sources that appear in more than 13 snapshots and show a lifespan of 13 appearances. This results in analysing 111,966 sources from 1,351 PLDs, a total of 46.59% of the total sources in our collection. Table 3.6 lists the top 10 PLDs and the number of URIs in the resulting corpus. If we compare this list with the top-10 PLDs in the original seed list(cf. Figure 3.4), we see that we have 5 overlapping PLDs which are all from the top-6 of the original seed URIs. To highlight differences in dynamicity between data providers, we later detail and compare results for the top-4 PLDs in Table 3.6, each with more than 3k sources.

Q1: HOW FAST DO RESOURCES CHANGE?

To answer this question, we compute and analyse the average change frequencies of the sources. Let us assume a source s changed 5 times during our monitoring interval of 15 weeks. In this case, we can estimate the average change frequency λ of s to be 15 weeks/5 changes = 3 weeks.

Following this example, we summarise results for the average change frequency of sources across all domains (see Figure 3.4a) and for each of the top-4 data providers (see Figure 3.4b). We observed that 60% of the monitored sources are static considering the change frequency across all data providers. This observa-



(a) All data providers. (b) Top 4 data providers.

Figure 3.4: Fraction of sources with given average change frequency.

tion is similar to the results we obtained in a preliminary previous study [Umbrich et al., 2010a]. We refer to the remaining 40% of the sources (46k) as *dynamic sources*. From Figure 3.4a we can observe that roughly half of these non-static sources have a content change rate of 1 week. A total of 8% of all sources changed only once in our experiment.

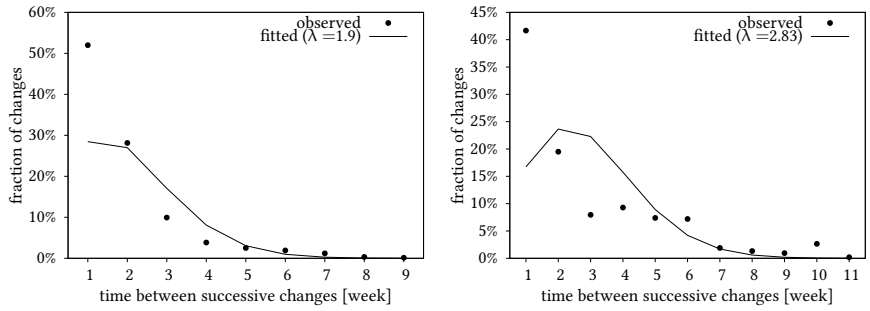
Looking at Figure 3.4b, we see the different change frequency distributions for the top-4 data providers. The `identi.ca` data provider – a social micro-blogging platform – shows similar characteristics to the overall trend in Figure 3.4a. Similar behaviour is shown by the `freebase.com` provider for which about 60% of the sources are either very static or with only one change and for which around 22% of the sources are highly dynamic with a change rate (potentially) more frequent than 1 week. The `europa.eu` data provider shows that we detected only one change for nearly 80% of the sources. A manual inspection showed that bulk changes happened in the 11th week of our experiment. For the `dbpedia.org` dataset, we observe that over 50% of the pages change every 6 or 7 weeks.

Q2: CAN THE CHANGES BE MODELLED AS A MATHEMATICAL PROCESS?

Next, we analyse whether we can apply the Poisson model to the changes of sources detected in our analysis. Various authors discovered that the *change behaviour* of Web pages corresponds closely with – and can be predicted using – a Poisson distribution [Brewington and Cybenko, 2000a,b; Cho and Garcia-Molina, 2003a,b]. Such knowledge can be exploited in various task, e.g., we can predict the next most likely document change and developing adaptive data update strategy for search engines. We use an established model for changes of Web documents and see if our observed changes can be also modelled accordingly.

Therefore, we compute the average change rate λ for each document. We group sources with the same change rate and compute and plot their distribution of successive change intervals; e.g., a document which changed in week 2 and 6 has a successive change interval of 4.

To answer our question, we performed a log-linear regression and used the maximum likelihood method to estimate the parameter λ for a Poisson distribution. We measured how well the estimated poisson distribution fits our observed data with a Chi square test (χ^2) [Vose and Vose, 2000]. Table 3.7 lists the observed and estimated value λ and the total number of change events. the results of the distribution fitting and the goodness of fit tests for the different classes of change frequencies. Even some of the estimated parameters are close to the observed average change frequency, non of the distribution could be fitted to a Poisson model with statisti-



(a) $\lambda = 2$ (b) $\lambda = 3$

Figure 3.5: Sources with an average change frequency of 2 (3.5a) and 3 weeks (3.5b).

avg. change frequency (λ)		
observed	*fitted*	*#events*
1	1.04	72,630
2	1.9	1,510
3	2.83	528
4	3.77	1,443
5	3.62	2,817
6	5.31	2,726
7	5.51	3,210
12	9.8	810
13	8.89	1,442
14	9.24	3,989

Table 3.7: Result estimates of the distribution fitting and Chi square (χ^2) test.

cal significancy: all p-values of the χ^2 test were less than 0.05 which indicates that some factor other than chance is the cause for the deviation.[6]

We selectively present the graphs for sources which appeared in 13 snapshots and show an average change frequency of 1 weeks (Figure 3.5a) and for 3 weeks (Figure 3.5b). The predicated poisson process is plotted in the graphs as the line together with the estimated value for λ, the average change frequency. If the changes can be modelled as a Poisson process, the resulting graph should be distributed exponentially.

So far, we observed different levels of dynamicity for Linked Data sources in our data collection. Moreover, the different change frequency distributions for the top-4 data providers suggest that the change processes are also data provider dependent.

3.2.2.1 *Change characteristics*

We are now interested in categorising the type of occurring changes and, as before, to see if the change behaviour varies across different data-providers.

Q3: CAN WE OBSERVE DIFFERENT TYPES OF CHANGES?

6 For example, a p-value of 0.01 means that there exists a 1% chance that the deviation between observed and estimated data is due to chance alone.

	Fraction				
CHANGE TYPE	all	identi.ca	freebase.com	dbpedia.org	europa.eu
UPDATE	24.20%	2.21%	11.71%	8.57%	–
UPDATE/MIXED	23.56%	0.33%	21.74%	20.85%	0.01%
ADD	20.59%	11.55%	50.28%	3.13%	99.99%
MIXED	19.03%	10.07%	1.64%	38.02%	–
DEL/ADD	2.65%	4.84%	6.16%	1.88%	–
ADD/MIXED	1.69%	16.52%	0.43%	5.56%	–
DEL/MIXED	1.65%	18.07%	0.13%	15.32%	–
DEL	1.16%	10.40%	0.23%	–	–
REST	5.47%	26.01%	7.68%	6.67%	–

Table 3.8: Results of source change categories and combination.

We now discuss the observed types of changes for the sources over time. Therefore, we defined the following four main change categories:

- ADD – we classify a change as *added* if only new statements are attached to a source;
- DEL – a change is categorised *deleted* if only existing statements are removed;
- UPDATE – we say a source is *updated* if the number of added and removed statements were the same and for each removed statement there exists an added statement which overlaps by two elements (e.g., the subject and object values are the same);
- MIXED – we use the category *mixed* when statements are added and deleted but the changes do not classify as an update.

We categorise the type of changes observed in the 15 weeks of the experiment for each of the 46k dynamic sources (cf. Figure 3.4a). For example, if only new content was added to a source between two snapshots and between two other snapshots one or more statements were just updated, we label the source with ADD/UPDATE. Similarly, we label a source with DEL/ADD if the detected change events were deletions and additions. Table 3.8 lists the fraction of sources which encountered such a change (or combination thereof) for each of the categories in descending order. We do not present the combination of change categories with less than 1% of the sources. In addition, we provided analogous figures for each of the top-4 data providers.

The study shows that the most frequent observed changes are purely updates or updates combined with some additional removals or new statements. In detail, we measured that for 25% of the dynamic sources, content changes were purely updates. Furthermore, we observed a combination of update events with mixed events for another 24% of the sources. Interestingly, 20% of the sources showed a monotonic behaviour such that in each change event, new statements are added and old statements remain as before (ie., not updated or deleted).

A closer look at the results for the top-4 data providers suggests that the variation of changes across publishers. The changes of the `identi.ca` sources show combinations of most of the categories which is explainable by the fact that users of this micro-blogging service do not only post new messages, but also update their profile information and follow or unfollow new and existing users. The most changes observed for the `freebase` provider are only additions of information to existing content or updates. The changes for the `dbpedia.org` provider are a

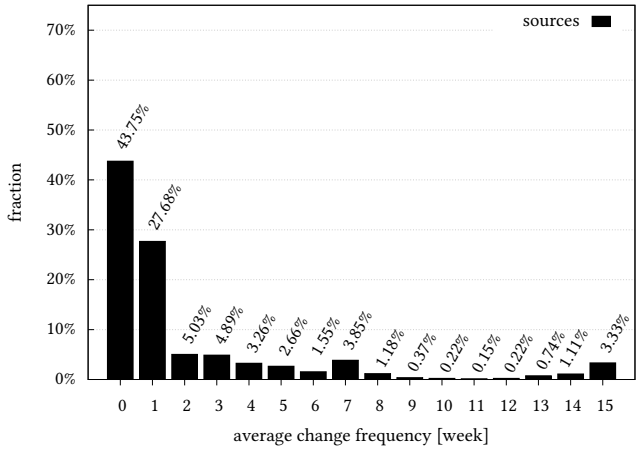


Figure 3.6: Fraction of PLDs with given average change frequency.

combination of adding and removing statements at the same time. Interestingly, the changes observed for the government data from `europa.eu` were almost all only additions of new statements. This verifies also our manual inspection of the changes of this data provider for the previous question. We can see that two of these four data providers show a high fraction of sources with a monotonic change behaviour, where only new information is added to existing data.

Overall, it seems like there is no clear categorisation of the changes for the sources. Most sources show a combination of different change categories. Most of the sources which have only additions (ADD) are dominated by `europa.eu` which seemed to have a single update for 80% of the sources in week 11 of our experiment.

Q4: CAN WE OBSERVE DIFFERENT CHARACTERISTICS FOR DIFFERENT DATA PROVIDERS?

By looking individually at the top-4 data providers, the previous questions already provide strong evidence that the change behaviour of the data sources depends on the data provider and the nature of the data, e.g., social micro-blogging, government data or encyclopaedias. Despite this, we still look into how the average change frequency of sources varies across different data providers. To do so, we group the sources based on their PLDs and computed the average change ratio. Figure 3.6 shows the distribution of PLDs with their specific average change frequency. We see that the around 43% of the data providers publish static data. The majority of the remaining data providers hosts sources which change on average every week. A total of 27% of the data providers publish data that changes on average at least once a week. The remaining 30% of the providers published data with an average change frequency varying between 2 and 15 weeks. This shows that we can observe different characteristics for different data providers.

3.3 INDEX FRESHNESS STUDY

In the previous section, we verified that Linked Data is dynamic and shows different levels of dynamicity across different data providers. The fact that the Web of Data is dynamic and very large poses serious challenges for maintaining a comprehensive and up-to-date index. Typically, the coverage of an index is limited by the amount of data that can be located, retrieved and indexed by local servers. Sim-

ilarly, the size and dynamicity of sources and the availability of local resources directly affects the ability of centralised query engines to keep the indexed data coherent with their counterparts on the Web which requires constantly monitoring the sources and reflecting changes in the index.

Thus coverage and freshness are crucial factors and have an impact on the results of the query processing. Particularly, if the users often require either up-to-date results and/or high recall for queries. In fact, several authors motivate their Linked Data query approaches based on the assumption that centralised indexes are incomplete and outdated [Hartig et al., 2009; Karnstedt et al., 2012; Ladwig and Tran, 2010; Li and Heflin, 2010; Stuckenschmidt et al., 2004; Tian et al., 2011]. However, to the best of our knowledge, there exists no study which quantifies these assumptions. Therefore, we designed an experiment which allows us to assess the freshness and coverage of a given store, which we refer to as the *coherence* of a store. We then, investigate two prominent public indexes for the Web of Data. Our obtained results verify that indexes for Linked Data in the Web are often incomplete and outdated.

In the following, we detail our methodology and critically discuss the obtained results.

3.3.1 Methodology

Our aim is to measure how coherent public SPARQL stores are with regards to current information on the Web. Thus, our methodology is based on SPARQL queries which we execute directly over the Web using the link traversal based query execution approach (cf. Section 2.3.3.1) and over two selected stores. We can draw conclusions about the index coverage and freshness by comparing the two obtained result sets.

PROBING THE COHERENCE OF SPARQL STORES In order to test the coherence of a given store, we propose to probe both the store and the Web with a broad range of simple SPARQL queries. We could perform the probing based on a source-centric perspective by comparing the data for a Web source against the data cached in the corresponding graph using `GRAPH` queries. However, (and as is the case for the two caches we test later) many stores do not have consistent naming of graphs: sometimes the graph may indeed refer to a particular Web source, but oftentimes the graph will be a high-level URI (e.g., `http://dbpedia.org`), informally indicating a dump from which the data were loaded but which cannot be directly retrieved. Moreover, some stores do not support `GRAPH` queries.

```
SELECT ?sIn, ?pIn, ?oOut, ?pOut
WHERE {
  ?sIn ?pIn <entityURI> .
  <entityURI> ?pOut ?oOut .
}
```

Query 3.1: Entity-query template

Therefore, we decided to perform our experiment based on a entity perspective and use the template query as shown in Query 3.1. This query returns all values of RDF triples in which the given entity URI appears in either the subject or object position.

Figure 3.7 illustrates how the results obtained by querying directly the Web (L) and results returned by the materialised cache (S) may diverge. We view results as consisting of sets of sets of variable bindings (i.e., $\mathsf{S}, \mathsf{L}, \Delta^* \subset 2^{\mathbf{V} \times \mathbf{UL}}$ reusing common notation for the set of all query variables, URIs and literals resp.); we ex-

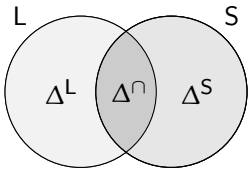


Figure 3.7: Result set (Venn) diagram

clude answers that involve blank nodes to avoid issues of scoping and inconsistent labelling between the results from the Web and the SPARQL store.

We reuse notation outlined in Figure 3.7. $\Delta^{\cap} := L \cap S$ refers to the set of results in both L and S, i.e., results for which the store is up-to-date. $\Delta^{L} := L \setminus S$ refers to the set of results returned by the Web that are not returned by the store. Some of these results may be caused by remote data changing which the store did not update; others may be from sources that the store did not cache. $\Delta^{S} := S \setminus L$ indicates results returned only by the store. Some of these results may be stale; others might be accurate but involve a source that the live engine did not access.

COHERENCE MEASURE We compute the coherence of a SPARQL store for a given query based on the number of results from the Web (L) and the number of overlapping and thus coherent results returned by the store ($\Delta^{\cap}$). More formally, let q be a instantiation of our template entity query, L_q the results from the Web with the LTBQE approach and S_q the results from a given store. We compute the overlapping results $\Delta^{\cap}_q = S_q \cap L_q$ to calculate the coherence value for the query q and the given store as the ratio of overlapping results vs. all results obtained from the Web, as follows:

$$\mathsf{coh}(q) = \frac{|\Delta^{\cap}_q|}{|L_q|}$$

Next, we present the two stores under study.

SELECTED PUBLIC SPARQL STORES The two selected public SPARQL stores for this study are the “LOD Cache” hosted by OpenLink, and “the Semantic Web Index" hosted by Sindice.

- The *LOD Cache* is perhaps the most comprehensive SPARQL store for the Web of Data with the highest coverage of the available datasets. The store covers most of the registered CKAN data sets, but also published RDF data from other providers and hosts, at the time of writing, adding up to over 50 billion RDF statements.
- *The Semantic Web Index* from the Sindice team collects its data by crawling the Web and accepting user submissions. Instead of focusing only on RDF documents, the index supports other formats such as RDFa, the open graph protocol or the schema.org catalog.

EVALUATION QUESTIONS Given our methodology, we aim to answer the following questions with our evaluation:

Q1: *How coherent are the two stores?*

One core assumption of this thesis is that materialised indexes which aim to cover large parts of the Web of Data cannot keep their index data consistent

with their Web sources. As such, we aim to validate this assumption with our study.

Q2: *Are coherence measures store independent?*

We highlighted that the two Linked Data SPARQL stores in this study are indexing different parts of Linked Data and very likely do so at different times. However, we also expect that there is a certain overlap of the indexed data. As such, we investigate if the two stores show different data coherence for the same data.

Q3: *Are coherence measures data provider independent?*

Thus far, we focused on the characteristics of the store considering single queries. For this question, we are interested to see if the coherence measures vary for different data providers.

Q4: *How diverse are coherence measures for each data provider?*

Eventually, we are curious as to how diverse the coherence measures are for entities having the same publisher.

3.3.2 Evaluation

For our experiment, we decided to probe the two selected public SPARQL stores with a broad range of queries. We also considered using the SPARQL 1.1 `SAMPLE` keyword to collect entity URIs for our experiment. However, the Virtuoso SPARQL store does not support SPARQL 1.1 (though it does support similar custom syntax) and both stores are powered by this engine. Further, the SPARQL 1.1 `SAMPLE` operator makes no guarantees about the randomness of results.

We thus instead sampled a diverse set of entities from the 2011 Billion Triple Challenge dataset to instantiate our template query with.

SAMPLING QUERIES We randomly sampled dereferenceable URIs from the 2011 Billion Triple Challenge dataset[7], which covers around 8 million RDF Web documents and was introduced earlier in this chapter. To guarantee a broad and diverse sample of data providers and and a equal distribution of entities across these data providers, we performed the following sampling steps:

1. we grouped the dereferenceable URIs per PLD,
2. we picked randomly a PLD and
3. we then picked randomly dereferenceable URI from the selected PLD.

We sampled a total of 12k entity URIs spanning 581 PLDs with a mean of 21 and a max of 86 entities per PLD.

EXECUTION The experiment is conducted as follows: Firstly, we created an instance of our entity query (cf. Query 3.1) for each of the 12k URI. Next, we use the Java library ARQ to execute the queries against both stores and store the results. We decided to execute the entity queries over the Web with our own LTBQE implementation, which we present and discuss later in Chapter 4. Again, we store the SPARQL results. Eventually, we compare the obtained results from the SPARQL store and LTBQE and compute the coherence measure for each query as described.

7 `http://challenge.semanticweb.org/`

	Web		Sindice		OpenLink	
successful queries	12,277		12,004		8,707	
non empty results	8,861	(72%)	4,241	(35%)	8,631	(99%)
	Sindice ∩ Web			OpenLink ∩ Web		
non empty results	3,090			6,289		

Table 3.9: Number of successful and non-empty queries for the two stores

	Sindice		OpenLink	
№	PLD	#queries	PLD	#queries
1	bestbuy.com	74	stanford.edu	60
2	tdwg.org	61	openlinksw.com	59
3	kanzaki.com	59	anl.gov	57
4	semanlink.net	58	openresearch.org	55
5	revyu.com	57	bestbuy.com	52

Table 3.10: Top-5 data providers based on the number of queries.

3.3.2.1 *Findings*

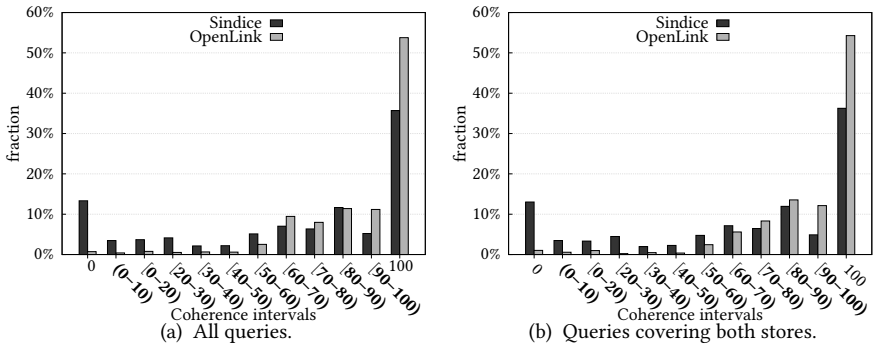
The following findings are obtained from an experiment which we ran in early March 2012. We successfully executed (without error) a total 12,277 entity queries over the Web, 12,004 (97%) against the Sindice index and 8,707 (71%) against the OpenLink store. The differences between the number of Web queries and index queries is due to the fact that some of the queries timed out or caused HTTP exception when executed over one of the two stores.

Table 3.9 shows that surprisingly, only 72% of the queries returned results if they are executed directly over the Web. This is surprising since all of the sampled URIs are dereferenceable according to the information provided in the BTC 2011 dataset. The OpenLink index return results for 99% of the successful queries, whereas the Sindice index covered only 33% of the queries. This result reflects the two slightly different focuses of the two stores. As we mentioned before, the OpenLink store aims to provide a comprehensive coverage of the Linked Open Data Web and indexes potentially all published data dumps, whereas the Sindice index focusses of a broad coverage of structured data on the Web as such, including various other formats such as Microformats.

Furthermore, we see from the results in the table, that we have 3k queries for which the results from the Web and the Sindice store were both non-empty, where 6k queries returned non-empty results if executed with LTBQE and against the OpenLink store. We believe that we get the most interesting results by comparing the non-empty result sets and thus, we focus in the remainder of this evaluation on these two sets of queries.

The 3k queries for the Sindice store span across 283 PLDs with a mean of 10 queries per PLD, and the 6k OpenLink queries cover 369 PLDs with a mean of 17 queries per PLD. Table 3.10 lists the top-5 data providers based on the number of non-empty queries in both query sets.

Q1: HOW COHERENT ARE THE TWO STORES



(a) All queries. (b) Queries covering both stores.

Figure 3.8: Coherence distribution.

For the two stores under analysis, Figure 3.8a illustrates the percentage of queries that fall into different intervals of coherence values. We decided to split the values into 10 intervals and separate the two extreme cases of full coherence (coh(q)=1) and respectively no coherence (coh(q)=0). We use square brackets to indicate that a value is included in an interval and parentheses of the value is not included in the interval. Thus the figure shows 12 intervals; the y-axis is in linear scale, and the linear x-axis intervals represent the coherence measures as percentages (the right of the graph indicates increasingly coherent queries). We can see that the OpenLink store returned fully coherent results for 56% of the queries, whereas the Sindice store returned fully coherent results for around 35% of the queries. We believe that OpenLink was extensively updated in Feb. 2012. In contrast, information for 1% of the tested queries in the OpenLink index are entirely missing or out-of-date ($\mathsf{coh}(q) = 0$), versus 15% for Sindice; these high percentages are due to partial coverage of Web sources and outdated data-dumps in the index.

Q2: ARE COHERENCE MEASURES STORE INDEPENDENT?

Next, we look if the coherence values for a given entity is independent from the store. We consider for this experiment only queries which returned non empty results for both stores and the LTQBE approach. In total we found 2,126 of such queries. Figure 3.8b shows again the coherence interval distribution for these queries. The distribution for the same queries differs between the both stores indicating that the coherence measures are store dependent. We also observe that the plot is very similar to the one shown in Figure 3.8a indicating that the selected queries are not a specific subset but rather reflect also the overall characteristic of a store.

Q3: ARE COHERENCE MEASURES DATA PROVIDER INDEPENDENT?

So far, we focused on the characteristics of the store considering single entities. For the next experiment, we are interested if the coherence measures vary for different data providers. Therefor, we grouped the entities based on their PLD and computed the average coherence for the group of entities. We plotted the average coherence distribution for the PLDs in Figure 3.9. Again, we can see a similarity to the coherence distribution on an entity level as in Figure 3.8a and Figure 3.8b. However, in contrast, we can see that the majority of the PLDs have a coherence between 90% and 100% for the OpenLink store. We show in Table 3.11 the average coherence value for our top-5 PLDs for both stores (cf. Table 3.10). All of the listed PLDs have an average coherence of more than 60% with a maximum of 97% for `kanzaki.com` and Sindice and 96% for `openlinksw.com` and OpenLink. In fact,

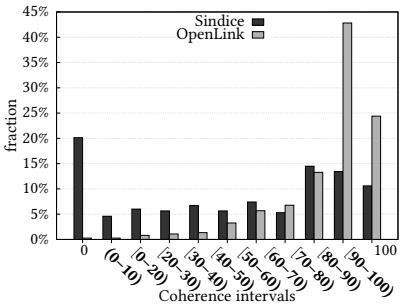


Figure 3.9: Average coherence distribution grouped by PLDs.

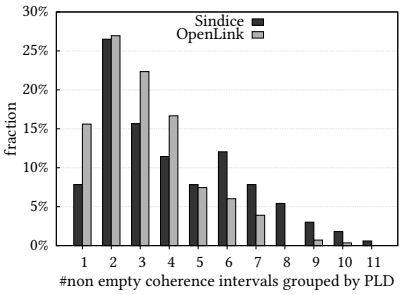


Figure 3.10: PLD Coherence variation.

№	Sindice		OpenLink	
1	bestbuy.com	77.19%	stanford.edu	67.85%
2	tdwg.org	60.16%	openlinksw.com	90.61%
3	kanzaki.com	97.27%	anl.gov	96.62%
4	semanlink.net	80.04%	openresearch.org	75.68%
5	revyu.com	87.40%	bestbuy.com	77.56%

Table 3.11: Top-5 data providers and their average coherence value.

the OpenLink store is hosted by the `openlinksw.com` provider which explains the high coherence.

Q4: HOW DIVERSE ARE COHERENCE MEASURES FOR EACH DATA PROVIDER? Eventually, we are interested how diverse the coherence measures are for entities having the same PLD. Therefor, we again grouped the entities by their PLD and counted in how many different coherence intervals we can group the entities. This gives us a value between 1 and 11 for each PLD. Let us assume that we tested two entities for a PLD and measured a coherence value of 0.5 for the first entity and 0.8 for the second query; this results in two groups. We excluded PLDs with only 1 query. Figure 3.10 shows the distribution of PLDs and the number of different coherence intervals for their entities. We can see that the entities of around 25% of the PLDs can be divided into two different intervals. Interestingly, we observe that the OpenLink store has more PLDs with up to four different intervals than the Sindice store. In contrast, the Sindice store indexed more PLDs than OpenLink for which the entities can be classified in more than 6 coherence intervals.

Eventually, we show the distribution of coherence intervals of the entities for the top-5 PLDs for both stores in Figure 3.11. The graphs explain in more detail the average values from Table 3.11. Let us inspect the PLD `bestbuy.com` more closely, since it is the only overlapping PLD in the top-5 of the two stores with a similar average coherence value of around 77%. We find that roughly 30% of the queries for both stores have a coherence value in the interval $[60 - 70)$. Moreover, the OpenLink store has another 28% of the queries in the $[90 - 100)$ interval, whereas the Sindice store maintained a coherent view for 28% of the remaining entity queries. This is perhaps an indication that Sindice has more recently updated parts of their `bestbuy.com` data through their crawling strategies, highlighting that the source collection methods affect coherence of different sources in different ways for different data.

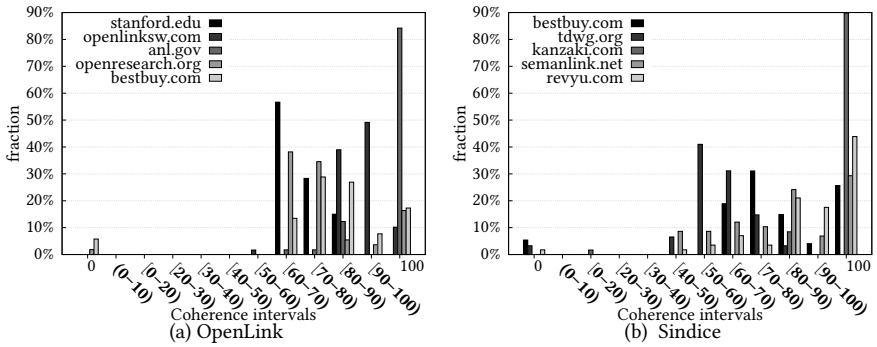


Figure 3.11: Distribution of coherence values for the top 5 PLDs per store.

3.4 CONCLUSION

In this chapter, we introduced the evaluation datasets for this thesis and provided validating evidences for the core assumptions that the Web of Data is continuously changing and that these dynamic processes pose insurmountable challenges for materialised SPARQL stores that aim to maintain coherent data coverage and up-to-date query results.

EVALUATION DATASETS We discussed in detail two quite divergent perspectives on the Web of Data and contrasted the weakness and strength of both views. The BTC dataset view consists of data which is crawled from the Web of Data for the annual Billion Triple Challenge at the International Semantic Web Conference (ISWC). The second view is the CKAN/LOD cloud repository which indexes publisher-reported statistics about datasets and guarantees that the listed datasets fulfil a set of certain criteria such as external links or a minimum number of statements.

We decided to combine both data sets to get the best possible coverage for our study of the evolutional process on the Web of Data. However, we will use the BTC datasets for the experiments presented in Chapter 4– Chapter 6 since the dataset covers more domains (791) than the CKAN/LOD, is empirically validated in several papers, and includes vocabularies and decentralised datasets. In addition, the dataset was crawled by traversing links, which is a very related approach to the link traversal based query processing that is central to this thesis. We will exploit the fact that the dataset contains dereferenceable URIs, millions of loosely connected sources and vocabularies hosted on the Web and we will use it to generate realistic and executable queries to evaluate our approaches directly over the Web, across a broad range of providers.

LINKED DATA IS DYNAMIC We described an experiment to study and quantify the dynamics on the Web of Data. To ensure a broad coverage of the data for this study, we use a core set of 95 k sources which are (i) randomly selected from the BTC dataset and (ii) from registered datasets in the CKAN/LOD catalog. We continue to monitor the sources in this kernel once a week, which allows us to study the evolution of the sources in a contiguous fashion and in addition, perform a crawl starting from the kernel sources which allows for studying the evolution of the neighbourhood sources.

At the time of writing, we had access to 15 weekly snapshots and the study validated our assumptions and further showed that:

- 40% of the sources which appeared in nearly all snapshots are dynamic and, interestingly, the content of 17% of the sources changed between two monitoring events;
- similar to a previous study in 2010 [Umbrich et al., 2010a], roughly 60% of the sources did not change during the period of our experiment;
- we cannot verify that the observed change processes of sources can be modelled and explained with the mathematical model of a Poisson process, which is the well studied and accepted model for changes in Web sources.

CHARACTERISTICS OF CHANGES We also classified the content change of a source based on removal or addition of statements or a combination of both. The resulting characterisation of sources based on the occurring change types showed that:

- 24% of the observed changes can be classified as updates for which only one node of a triple changed (e.g., one reading from a sensor or a change of a social network profile picture). This is particularly interesting from a query perspective since the knowledge of such change characteristics can be exploited as we will show in Chapter 6;
- 20% of the dynamic sources increase the number of triples with each change and thus show a monotonic change behaviour.

Moreover, we grouped the sources by the PLD of the data providers and observed that the average change frequency of sources varies between data providers. In addition, we inspected the sources of the top-4 biggest data provider in our dataset to highlight differences in their change behaviour and also in the change characteristics of the sources across data providers.

As a side remark, we did not fully exploit the possibility to study the neighbourhood growth rate of the Web of Data with our monitoring experiment. However, we recognised from our analysis that for at least 20% of the sources, only new statements were added during a change events which gives an indication that more links are created. Although this is a very interesting and relevant area of study but it is not in the scope of this work, where we focus on initially verifying the general dynamicity of Linked Data.

CENTRALISED INDEXES ARE PARTIALLY COHERENT WITH THE WEB Our second experiment was designed to validate the assumption that materialisation approaches potentially return outdated or incomplete results due to index maintenance problems with respect to updates and coverage. Our study is the first to compare query results obtained from the Web with results from a SPARQL index and establishes a measure of index coherence for an store based on simple template queries. We randomly sampled 12k entity queries from the BTC dataset and compared for each query the results returned by executing the query against the Web and against the two prominent public SPARQL stores of the Sindice and OpenLink team. We measured that over 60% of the query results obtained from the Sindice index did not entirely cover the results returned from the Web with the LTBQE approach. For the remaining 40% of the queries, the returned results from the index were coherent with the results from the Web. The OpenLink index showed similar characteristics with 60% coherent results and 40% partially coherent results.

An interesting observation is the similarity of the approximate 60% coherent results of the OpenLink index with the 60% static sources in our data dynamics experiment. This fact (and the observed 60% static sources in a former experiment)

can be seen as an initial indication that the Web of Data has around 60% static content. However, one would need to scale up our experiments in terms of number of queries, sources and duration to provide stronger evidences.

In summary, due to the dynamicity of Linked Data, we observed that traditional materialisation-based query approaches potentially return missing or even false results for SPARQL queries. Thus, we address this problem by systematically investigating alternative query approaches for Linked Data in the remainder of this thesis. In the next chapter, we begin by studying how the link traversal based query execution approach performs in practice and propose extensions to overcome some weaknesses.

4 ON LINK TRAVERSAL QUERYING FOR A DIVERSE WEB OF DATA

The results from the previous chapter confirm the necessity for alternative query approaches to centralised indexes if the up-to-dateness of results is a desired requirement. Exploiting the Linked Data principles allows for answering queries "live" over the Web by dereferencing URIs and traversing remote data sources at runtime. These *link traversal based query execution* (LTBQE) approaches for Linked Data offer up-to-date results and decentralised (*i.e.*, client-side) execution, but must operate over incomplete dereferenceable knowledge only available in remote documents, thus affecting response times and "recall" for query answers.

In this chapter, we systematically study the recall and effectiveness of LTBQE in practice for Linked Data. To help bridge data heterogeneity in diverse sources, we propose lightweight reasoning extensions to find additional answers. Starting from the state-of-the-art which considers only dereferenceable information and follows `rdfs:seeAlso` links [Hartig et al., 2009], we propose extensions to consider first `owl:sameAs` links and reasoning, and second lightweight RDFS reasoning.

The chapter is organised as follows:

- In Section 4.3, we discuss and exemplify some of the limitations of the LTBQE approach regarding the expected result recall, which not only depends on the input query but also on query planning, dereferenceability of URIs and the interlinkage of the remote data.
- Next, in Section 4.2, we introduce LiDaQ (Linked Data Query engine): our implementation of LTBQE using the iterator-based model of SQUIN [Hartig et al., 2009], but also features novel reasoning extensions and optimisations.
- We empirically estimate the data recall of link traversal query techniques and how much more raw dereferenceable data our proposed extensions make available in practice by analysing the BTC'11 crawl in Section 4.3.
- In Section 4.4, we review current SPARQL benchmarks if they can be run against real-world Linked Data sources, survey the published evaluation of related approaches and consequently address the shortcomings with a novel benchmark methodology, called QWalk.
- We test LTBQE and its extensions for three query benchmarks in a realistic, uncontrolled setting and critically discuss our obtained measures in Section 4.5.
- Section 4.6 concludes with final remarks.

4.1 ON THE (IN)COMPLETENESS OF LTBQE

An open question for link traversal based query execution approaches is the *decidability* of collecting query-relevant sources: does the LTBQE approach always terminate? This is dependent on whether one considers the Web of Data to be infinite or finite. For an infinite Web of Data, this process is indeed undecidable [Hartig,

2012]. To illustrate this case, Hartig [2012] uses the example of a Linked Data server describing all natural numbers[1], where each $n \in \mathbb{N}$ is given a dereferenceable URI, each n has a link to $n+1$ with the predicate `ex:next`, and a query with the pattern "`?n ex:next ?np1 .`" is given. In this case, the traversal of query-relevant sources will span the set of all natural numbers. However, if the (potential) Web of Data is finite, then LTBQE is decidable; in theory, it will terminate after processing all sources. The question of whether the Web (of Data) is infinite or not comes down to whether the set of URIs is infinite or not: though they may be infinite in theory [Berners-Lee et al., 2005] (individual URIs have no upper bound for length), they are finite in practice (machines can only process URIs up to some fixed length).[2]

Of course, this is a somewhat academic distinction. In practice, the Web of Data is sufficiently large that LTBQE may end up traversing an unfeasibly large number of documents before terminating. A simple worst case would be a query with a "open pattern" consisting of three variables as in the following example.

Example 4.1. The following Query 4.1 asks for a general description of people known by `oh:olaf`:

```
SELECT ?s ?p ?o
WHERE {
  oh:olaf foaf:knows ?s .
  ?s ?p ?o .
}
```

Query 4.1: General description of friends

The first query-relevant sources will be identified as the documents dereferenced from `oh:olaf` and `foaf:knows`. Thereafter, all triples in these documents will match the open pattern, and thus all URIs in these documents will be considered as potential query-relevant links. This will continue recursively, crawling the entire Web of Data.

Of course, this problem does not occur only for open patterns with three variables but can also happen for more restrictive query patterns, as our next example shows.

Example 4.2. One could also consider the following Query 4.2 which asks for the friends-of-friends of `oh:olaf`:

```
SELECT ?o
WHERE {
  oh:olaf foaf:knows ?s .
  ?s foaf:knows ?o .
}
```

Query 4.2: Friends of Friends

This would end up crawling the connected Web of FOAF documents, as are linked together by dereferenceable `foaf:knows` links.

Partly addressing this problem, Hartig et al. [2009] defined an iterator-based execution model for LTBQE, which rather approximates the answers provided

1 Such a server has been made available by Vrandečić et al. [2010], but unfortunately stops just shy of a billion. See, *e.g.*, `http://km.aifb.kit.edu/projects/numbers/web/n42`.

2 It is not clear if URIs are (theoretically) finite strings. If so, they are countable [Hartig, 2012].

by Definition 8. This execution model defines an ordering of triple patterns in the query, similar to standard nested-loop join evaluation. The most selective patterns (those expected to return the fewest bindings) are executed first and initial bindings are propagated to bindings further up the tree. Crucially, later triple patterns are partially bound when looking for query-relevant sources. Thus, taking the previous example, the pattern "`?s foaf:knows ?o .`" will never be used to find query-relevant sources, but rather partially-bound patterns like "`cb:chris foaf:knows ?o .`" will be used. As such, instead of retrieving all possible query-relevant sources, the iterator-based execution model uses interim results to apply a more focused traversal of the Web of Data. This also makes the iterator-based implementation order-dependent: results may vary depending on which patterns are executed first and thus answers may be missed. However, it does solve the problem of traversing too many sources when low-selectivity patterns are present in the query.

Whether defined in an order-dependent or order-independent fashion, LTBQE will often not return complete answers with respect to the Web of Data [Hartig, 2012]. We now enumerate some of the potential reasons LTBQE can miss answers.

NO DEREFERENCEABLE QUERY URIS: The LTBQE approach cannot return results in cases where the query does not contain dereferenceable URIs.

Example 4.3. For example, consider posing the following Query 4.3 against our example data graph in Figure 2.1, asking for all information for the URI `cb:chris`:

```
SELECT *
WHERE {
  cb:chris ?p ?o .
}
```

Query 4.3: List all information for a subject URI

As previously explained, the URI `cb:chris` is unfortunately not dereferenceable ($\mathsf{deref}(\texttt{cb:chris}) = \emptyset$) and thus, the query processor cannot compute and select relevant sources from interim results.

UNCONNECTED QUERY-RELEVANT DOCUMENTS: Similar to the previous case of reachability, the number of results might be affected if query relevant documents cannot be reached. This is the case if answers are "connected" by literals, blank-nodes or non-dereferenceable URIs. In such situations, the query engine cannot discover and dereference further query relevant data.

Example 4.4. The following Query 4.4 illustrates such a case where two query patterns are joined by literal values:

```
SELECT ?olaf ?name
WHERE {
  oh:olaf foaf:name ?name .
  ?olaf foaf:name ?name .
}
```

Query 4.4: List people with same name

Answers (other than `oh:olaf`) cannot be reached from the starting URI `oh:olaf` because the relevant documents are connected by the literal `"Olaf Hartig"`.

DEREFERENCING PARTIAL INFORMATION: In the general case, the effectiveness of LTBQE is heavily dependent on the amount of data returned by the $\mathsf{deref}(u)$ function. In an ideal case, dereferencing a URI u would return all triples mentioning u on the Web of Data. However, this is not always the case.

Example 4.5. For example, let us consider the following Query 4.5:

```
SELECT ?s
WHERE {
  ?s owl:sameAs dblpA:Olaf_Hartig .
}
```

Query 4.5: List all `owl:sameAs` Information

This simple query cannot be answered since the triple "`oh:olaf owl:sameAs dblpA:Olaf_Hartig .`" is not accessible by dereferencing `dblpA:Olaf_Hartig`.

The assumption that all RDF available on the Web of Data about a URI u can be collected by dereferencing u is clearly idealised; hence, later in Section 4.3 we will empirically analyse how much the assumption holds in practice, giving insights into the potential recall of LTBQE on an infrastructural level.

4.2 LIDAQ: EXTENDING LTBQE WITH REASONING

We now present the details of LiDaQ: our proposal of a Linked Data Query engine to extend the baseline LTBQE approach with components that leverage lightweight RDFS and `owl:sameAs` reasoning in order to improve recall. We first describe the extensions we propose (Section 4.2.1), and then describe our implementation of the system (Section 4.2.2).

4.2.1 LTBQE Extensions

Partly addressing some of the shortcomings of the LTBQE approach in terms of completeness (or, perhaps more fittingly, *recall*), Hartig et al. [2009] proposed an extension of the set of query relevant sources to consider `rdfs:seeAlso` links, which sometimes overcomes the issue of URIs not being dereferenceable (as per `cb:chris` in our example). In the LiDaQ system, we include this extension, and on top, we propose further novel extensions that apply reasoning over query-relevant sources to squeeze additional answers, which in turn may lead to recursively finding additional data sources.

First, we propose following `owl:sameAs` links, which, in a Linked Data environment, are used to state that more information about the given resource can be found elsewhere under the target URI. Thus, to fully leverage `owl:sameAs` information, we first propose to follow relevant `owl:sameAs` links when gathering query-relevant sources. Subsequently, we apply `owl:sameAs` reasoning, which supports the semantics of *replacement* for equality, meaning that information about equiva-

lent resources is mapped to all available identifiers and made available for query answering.

Second, we propose some lightweight RDFS reasoning, which exploits schema-level information from pertinent vocabularies and ontologies that describe the semantics of class and property terms used in the query-relevant data and uses it to infer new knowledge. In a first step, we have to make schema data available to the query engine, where we propose three mechanisms:

1. a *static collection* of schema data are made available as input to the engine;
2. the properties and classes mentioned in the query-relevant sources are dereferenced to *dynamically build a direct collection of schema data*; and
3. the direct collection of schema data is expanded by *recursively following links on a schema level*.

Using the schema data collected by one of these methods, in the second step, we apply rule-based RDFS reasoning to materialise inferences and make them available for query-answering.

We now describe, formally define and provide motivating examples for each of the three extensions:

1. following `rdfs:seeAlso` links,
2. following `owl:sameAs` links and
3. applying equivalence inferencing, and collecting schema information for applying RDFS reasoning.

4.2.1.1 *Following `rdfs:seeAlso` Links:*

First, let us motivate and give the intuition for the legacy `rdfs:seeAlso` extension with a simple example.

Example 4.6. Consider executing the following simple query (cf. Query 4.6) – which asks for images of the friends of `oh:olaf` – against the data in Figure 2.1 using baseline LTBQE methods:

```
SELECT ?f ?img
WHERE {
  oh:olaf foaf:knows ?f .
  ?f foaf:depiction ?img .
}
```

Query 4.6: Query requiring `rdfs:seeAlso` information

The query processor evaluates this query by dereferencing the content of the query URI `oh:olaf`, following and dereferencing all URI bindings for the variable `?f` and matching the second query pattern "`?f foaf:depiction ?n .`" over the retrieved content to find the pictures. However, the query processor needs to follow the `rdfs:seeAlso` link from `cb:chris` to `cbDoc:` since the URI `cb:chris` is not dereferenceable (recall that a dashed arrow in Figure 2.1 denotes dereferenceability). In summary, in this example, to find the document `cbDoc:` and ultimately answer the query, LTBQE needs to be extended to follow `rdfs:seeAlso` links.

As such, Hartig et al. [2009] proposed an extension of LTBQE to follow `rdfs:seeAlso` when looking for query-relevant sources. We briefly formalise this extension:

Definition 9 (LTBQE Extension 1: `rdfs:seeAlso`). *Reusing notation from Definition 8, given a dataset* Γ *and a set of URIs* $\mathcal{U}$*, first define:*

$$\mathsf{seeAlso}(\Gamma, \mathcal{U}) := \{v \in \mathbf{U} \mid \exists u \in \mathcal{U} \text{ s.t. } (u, \texttt{rdfs:seeAlso}, v) \in \mathsf{merge}(\Gamma)\}$$

Now, for a given query, expand the set of query relevant sources as follows:

$$\mathsf{qrel}'(Q, \Gamma) := \mathsf{qrel}(Q, \Gamma) \cup \mathsf{seeAlso}\big(\Gamma, \mathsf{qrel}(Q, \Gamma)\big)$$

The rest of the definition follows from Definition 8 by replacing $\mathsf{qrel}(.)$ *with the extended function* $\mathsf{qrel}'(.)$.

4.2.1.2 *Following and Reasoning over `owl:sameAs` Links:*

We next formalise and describe our novel extension for following `owl:sameAs` links and applying equality reasoning. We again begin with a motivating example to cover the intuition.

Example 4.7. Consider Query 4.7, asking for friends of `oh:olaf` that are also co-authors.

```
SELECT ?f
WHERE {
  oh:olaf foaf:knows ?f .
  ?pub dc:creator ?f .
  ?pub dc:creator oh:olaf .
}
```

Query 4.7: Query requiring `owl:sameAs` information

Executing this query over the data in Figure 2.1 using the baseline LTBQE approach will not return any answers, since explicit equality information about URIs is not supported. The `owl:sameAs` relationship between `oh:olaf` and `dblpA:Olaf_Hartig` states that both URIs are equivalent and referring to the same real world entity, and hence that the information for one applies to the other. In summary, to answer this query, LTBQE must be extended to follow `owl:sameAs` links and apply reasoning to materialise inferences with respect to the semantics of replacement.

We now formalise the details of this novel extension.

Definition 10 (LTBQE Extension 2: `owl:sameAs`). *We propose an extension of LTBQE to consider* `owl:sameAs` *links and inferences. First, given a set of URIs* $\mathcal{U}$ *and a dataset* Γ*, as before, define:*

$$\mathsf{sameAs}(\Gamma, \mathcal{U}) := \{v \in \mathbf{U} \mid \exists u \in \mathcal{U} \text{ s.t. } (u, \texttt{owl:sameAs}, v) \vee (v, \texttt{owl:sameAs}, u) \in \mathsf{merge}(\Gamma)\}$$

And define by extension:

$$\mathsf{qrel}''(Q, \Gamma) := \mathsf{qrel}(Q, \Gamma) \cup \mathsf{sameAs}(\Gamma, \mathsf{qrel}(Q, \Gamma))$$

By replacing $\mathsf{qrel}(.)$ *with* $\mathsf{qrel}''(.)$*, this latter function is used to find addition query-relevant sources by analogue to Definition 8.*

Now we must define the role of inference. Let R *denote the set of rules of the form* EQ-* in Table 2.1. Extending the notation from Definition 8, define:

$$'\Gamma_i^Q := \Gamma_i^Q \bullet \mathsf{R}$$

In other words, $'\Gamma_i^Q$ denotes the closure of the raw Γ_i^Q dataset with respect to `owl:sameAs` data. The full `owl:sameAs` extension is then described by replacing $\mathsf{qrel}(.)$ with $\mathsf{qrel}''(.)$ (to follow `owl:sameAs` links) and Γ_i^Q with $'\Gamma_i^Q$ (for all i; to apply inferencing at each stage) in Definition 8.

4.2.1.3 *Incorporating RDFS Schemata and Reasoning*

Finally, we formalise and describe our novel extension for retrieving and reasoning with respect to RDFS descriptions of classes and properties used in the query-relevant data. We again start with a motivating example.

Example 4.8. Take the following Query 4.8, which asks for the images(s) depicting `oh:olaf`:

```
SELECT ?d
WHERE {
  oh:olaf foaf:depiction ?d .
}
```

Query 4.8: `foaf:depiction` Query requiring RDFS schemata and reasoning

From Figure 2.2, we know that `foaf:depiction` is a sub-property of `foaf:img`, and we would thus hope to get the answer `"Olaf Hartig"`. However, returning this answer requires two thing: (i) retrieving the RDFS definitions of the FOAF vocabulary; and (ii) performing reasoning using the first four rules in Table 2.1. In this case, finding the relevant schema information (the first step) is quite straightforward and can be done directly at runtime since the relevant terms (`foaf:img` and `foaf:depiction`) are within the same namespace and are described by the same dereferenceable document. However, consider instead Query 4.9:

```
SELECT ?d
WHERE {
  oh:olaf rdfs:label ?d .
}
```

Query 4.9: `rdfs:label` Query requiring RDFS schemata and reasoning

In this case, we know from the FOAF schema that `foaf:name` is a sub-property of `rdfs:label`, and so `"Olaf Hartig"` should be an answer. However, no FOAF vocabulary term is mentioned in the query, and so the FOAF schema will not be in the query-relevant scope. To overcome this, we can provide a static set of schema information to the query engine as input, or we can dereference property and class terms mentioned in the query-relevant data to dynamically retrieve the relevant definitions at runtime.

Definition 11 (LTBQE Extension 3: RDFS)**.** *We propose an extension of LTBQE to consider RDFS schema data and inferences. Let Ψ denote an auxiliary Linked Dataset that contains some schema data. Let R denote rules* {PRP-SPO1, PRP-DOM, PRP-RNG, CAX-SCO} *in Table 2.1 (other finite RDFS rules can be added as necessary). Extending the notation from Definition 8, define:*

$$''\Gamma_i^Q := \Gamma_i^Q \cup \Psi_i \bullet \mathsf{R}$$

In other words, $''\Gamma_i^Q$ denotes the closure of the raw Γ_i^Q dataset and the schema data in Ψ_i at each stage (the subscript on Ψ_i indicates that schema data can be collected recursively, on the fly). The RDFS extension is then described by replacing Γ_i^Q with $''\Gamma_i^Q$ (for all i; to apply inferencing at each stage) in Definition 8.

We are then left to describe how Ψ_i may be acquired, where we provide three options (Ψ_i^{1-3}).

1. *A static corpus of schema data Ψ can be provided as input, such that $\Psi_i^1 \coloneqq \Psi$.*

2. *The class and property terms used to describe $''\Gamma_i^Q$ can be dereferenced. Letting $\mathsf{preds}(\Gamma)$ denote the set of all URIs mentioned in the predicate position of some triple in the merge of Γ, and letting $\mathsf{o\text{-}type}(\Gamma)$ denote the set of all URIs mentioned in the object position of a triple in the merge of Γ whose predicate is `rdf:type`, we can define Ψ_i^2 as:*

$$\Psi_i^2 \coloneqq \mathsf{derefs}\big(\mathsf{preds}(\Gamma_i^Q) \cup \mathsf{o\text{-}type}(\Gamma_i^Q)\big)$$

   or, in other words, the schema data obtained by directly dereferencing all predicates and values for `rdf:type` mentioned in the query-relevant data thus far.

3. *Finally, it is possible to extend the schema data by following schema-level links. For a Linked Dataset Γ, let $\mathsf{sl}(\Gamma)$ be a "schema links" function which extracts the set of all URIs u such that u is the subject or object of some triple $t \in \mathsf{merge}(\Gamma)$ with predicate `rdfs:subPropertyOf`, `rdfs:subClassOf`, `rdfs:domain` or `rdfs:range`; or u is the object of some triple $t \in \mathsf{merge}(\Gamma)$ with `owl:imports` as predicate. Now, taking Ψ_i^2 as above, let $\Psi_{i,0}^3 \coloneqq \Psi_i^2$, and thereafter, for $j \in \mathbb{N}$ define:*

$$\Psi_{i,j+1}^3 \coloneqq \mathsf{derefs}\big(\mathsf{sl}(\Psi_{i,j}^3)\big) \cup \Psi_{i,j}^3$$

   such that links are recursively followed up to a fixpoint: the least j such that $\Psi_{i,j}^3 = \Psi_{i,j+1}^3$. This fixpoint then represents Ψ_i^3. In other words, the third method of collecting schema extends upon the second method by recursively following core RDFS links and `owl:imports` links from the direct schema data.

The second and third methods involve dynamically collecting schemata at runtime. The third method of schema-collection is potentially problematic in that it recursively follows links, and may end up collecting a large amount of schema documents (a behaviour we encounter in evaluation later). However, where, for example, class or property hierarchies are split across multiple schema documents, this recursive process is required to "recreate" the full hierarchy.

All of our three extensions – following `rdfs:seeAlso` links, following `owl:sameAs` links and applying `owl:sameAs` reasoning, retrieving RDFS data (using one of three approaches) & applying RDFS reasoning – can be combined in a straightforward manner. In fact, some answers may only be possible through the combination of all extensions. We will later explore the effects of combining all extensions in Section 4.5.

4.2.2 LiDaQ Implementation

In order to build the LiDaQ prototype, we have re-implemented Hartig et al.'s iterator-based algorithm for LTBQE [Hartig et al., 2009] together with some optimisations such as a local per-query cache with an in-memory quad store [Hartig and Huber, 2011] and a more efficient join operator [Ladwig and Tran, 2011]. We then add our various extensions. The code-base is written in Java. Our architecture – depicted in Figure 4.1 – features five main components:

QUERY PROCESSOR: uses the Java library ARQ to parse and process SPARQL queries and format the output results.[3]

SOURCE SELECTOR: decides which query and solution URIs should be dereferenced and which links should be followed.

SOURCE LOOKUP: an adapted version of the LDSpider crawling framework performs the live Linked Data lookups required for LTBQE. LDSpider respects the `robots.txt` policy, blacklists typical non-RDF URI patterns (*e.g.*, `.jpeg`) and enforces a half-second delay between two consequential lookups for URIs hosted at the same domain to avoid hammering remote servers.[4]

LOCAL REPOSITORY: a custom implementation of an in-memory quad store is used to cache the content of all query relevant data, including inferences, as well as indexing triple patterns from the query to match against the data. Triple patterns are matched in a continuous fashion as new content is pushed to the cache, feeding the iterators. When reasoning is not enabled, the triple pattern index filters non-matching triples and discards them. This repository is also used to store static schema data, where needed.

REASONER: the Java-based SAOR reasoner is used to support the aforementioned rule-based reasoning extensions [Bonatti et al., 2011] (cf. Section 2.2.3), and can execute inferences over the new content as it arrives in conjunction with the cache.

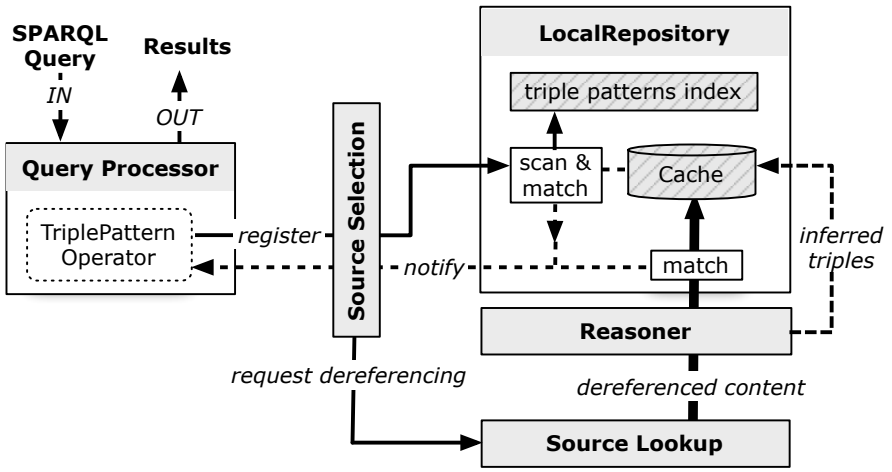


Figure 4.1: LIDAQ architecture diagram.

We further investigate a "reduced source-selection" variant of LIDAQ that uses straightforward, generic optimisations to minimise the number of sources considered while maximising results. Primarily, we avoid dereferencing URIs that do not appear in join positions: these are variables which are not used elsewhere in the query, but whose existence needs to be asserted. We illustrate this with a simple example:

Example 4.9. Consider the following query issued against the example graph of Figure 2.1, asking for friends of `oh:olaf` that have some value defined for `foaf:based_near`:

3 http://jena.sourceforge.net/ARQ/
4 http://code.google.com/p/ldspider/

```
SELECT ?f ?fn ?b
WHERE {
  oh:olaf foaf:knows ?f .
  ?f foaf:name ?fn .
  ?f foaf:based_near ?b .
}
```

Query 4.10: Location and name of friends

Assuming the `rdfs:seeAlso` extension is enabled, this query will first visit `ohDoc:`, then bind `cb:chris` for `?f`, and then visit `cbDoc:` for more information about `cb:chris`. Here, "`Chris Bizer`" will be bound for `?fn`, and `dbpedia:Berlin` bound for `?b`. However, dereferencing the latter URI would be pointless: we do not need any information about `dbpedia:Berlin` to answer the query. Here the variable `?b` does not appear in other triple patterns and further information about URIs bound to it are not directly required by the query. Our optimisation proposes to avoid wasting lookups up not dereferencing URIs bound to non-join variables.

Of course, by reducing the amount of sources and raw data that are accessed – and given that anyone in principle say anything, anywhere – we may also reduce the number of answers that are returned. Taking the previous example, for all we know, the document dereferenced through `dbpedia:Berlin` may contain (unrelated) information about friends of `oh:olaf`, which help to contribute other answers. However, we deem this to be unlikely in the general case, and note that it goes against the core LTBQE idea of using Linked Data principles to find query-relevant sources.

Aside from this optimisation to avoid dereferencing URIs bound to non-join variables, we posit that for real-world Linked Data, URIs in certain positions of a triple pattern may not be worth dereferencing to look for matching information. For example, given the pattern "`?s foaf:knows ?o .`", we would not expect to find (m)any triples matching this pattern in the document dereferenced by `foaf:knows`. In the next section, we investigate precisely this matter for different triple positions, and thereafter propose a further variation on LiDaQ's source selection to prune remote lookups that are unlikely to contribute answers.

4.3 EMPIRICAL STUDY

We now begin to look at how LTBQE and its extensions can be expected to perform in practice. In Section 2.3.3.1, we mentioned that the recall of the LTBQE approach is – in the general case – dependent on the dereferenceability of data. Along these lines, we can draw general conclusions about the effectiveness of LTBQE for answering queries over the Web of Data by looking at the nature of dereferenceability on the Web of Data. In particular, we are interested to know the ratio of information about available in dereferenceable sources versus the information available on the rest of the Web of Data. Or in other words, given a URI u, we want to know how much triples which mention u can we retrieve by dereferencing u and compare this number with the total number of triples mentioning u on the Web of Data. This gives us a indication as to what percentage of raw data is available to LTBQE versus, *e.g.*, a materialised approach with a complete index over a large crawl of the Web of Data. We can also similarly test how much additional raw information is made available by our extensions.

In summary, we take a large crawl of the Web of Data as a sample. We survey the ratio of all triples mentioning a URI in our corpus against those returned in

the dereferenceable document of that URI; we do so for different triple positions. We also look at the comparative amount of raw data about individual resources considering (1) explicit, dereferenceable information; (2) including `rdfs:seeAlso` links [Hartig et al., 2009]; (3) including `owl:sameAs` links and inferences; (4) including RDFS inferences with respect to a static schema. Unfortunately, we cannot study how much more raw data is made available by RDFS reasoning if the necessary schema data is dynamically imported. Our dataset for this analysis does not contain all the necessary information about the dereferenceability for all appearing schema data.

4.3.1 Empirical Corpus

For our corpus, we take the dataset crawled for the Billion Triple Challenge 2011 (BTC'11) in mid-May 2011 and which we discussed in detail in Section 3.1. We quickly recapture here the important features of the BTC'11 dataset for our analysis.

The corpus consists of 7.4 million RDF/XML documents spanning 791 pay-level domains (data providers). URIs extracted from all RDF triples positions, but without common non-RDF/XML extensions like `.pdf`, `.jpg`, `.html`, *etc.*, were considered for crawling. The resulting corpus contains 2.15 billion quadruples (1.97 billion unique triples) mentioning 538 million RDF terms, of which 52 million (~10%) are literals, 382 million (~71%) are blank nodes, and 103 million (~19%) are URIs. We denote the corpus here as $\Gamma_{\sim}$. The bulk of RDF data is serialised as N-Quads [Cyganiak et al., 2008], which provides information regarding which triple came from which document in a manner directly analogous to our notion of a Linked Dataset.

Alongside the bulk of RDF data, all relevant HTTP information, such as response codes, redirects, *etc.*, are made available. However, being an incomplete crawl, not all URIs mentioned in the data were looked up. As such, we only have knowledge of redir and deref functions for 18.65 million URIs; all of these URIs are HTTP and do not have non-RDF file-extensions. We denote these URIs by $\mathsf{U}_{\sim}$. Of the 18.65 million, 8.37 million (~45%) dereferenced to RDF; we denote these by $\mathsf{D}_{\sim}$.

Again, this corpus is only a *sample* of the Web of Data: we can only analyse the HTTP lookups and the RDF data provided for the corpus. Indeed, a weakness of our analysis is that the BTC'11 dataset only considers dereferenceable RDF/XML documents and not other syntaxes like RDFa or Turtle. Thus, with respect to the Web of Data, our high-level recall measures for LTBQE specify an *upper* bound: more information may be available on the Web of Data than we know about in our sample.

4.3.2 Static Schema Data

For the purposes of this analysis, we extracted a static set of schema data for the RDFS reasoning. As argued in [Bonatti et al., 2011], schema data on the Web is often noisy, where third-party publishers "redefine" popular terms outside of their namespace; for example, one document defines nine *properties* as the domain of `rdf:type`, which would have a drastic effect on our reasoning.[5] Thus, we perform authoritative reasoning, which conservatively discards certain third-party schema axioms (cf. [Bonatti et al., 2011]). In effect, our schema data only includes triples of the following form:

5 *viz.* `http://www.eiao.net/rdf/1.0`

Category	Triples	PLDs
`rdfs:subPropertyOf`	10,902	67
`rdfs:subClassOf`	334,084	82
`rdfs:domain`	26,207	79
`rdfs:range`	26,204	77
total	397,397	98

Table 4.1: Breakdown of authoritative schema triples extracted from the corpus

$$\begin{aligned}
\textsc{prp-spo1} &: (s, \texttt{rdfs:subPropertyOf}, o) \in \mathsf{deref}(s)\\
\textsc{prp-dom} &: (s, \texttt{rdfs:domain}, o) \in \mathsf{deref}(s)\\
\textsc{prp-rng} &: (s, \texttt{rdfs:range}, o) \in \mathsf{deref}(s)\\
\textsc{cax-sco} &: (s, \texttt{rdfs:subClassOf}, o) \in \mathsf{deref}(s)
\end{aligned}$$

We call these *authoritative schema triples*. Table 4.1 gives a breakdown of the counts of triples of this form extracted from the dataset, and how many domains (PLDs) they were sourced from: a total of 397 thousand triples were extracted from data provided by 98 PLDs. We denote this dataset as $\Psi_\sim$.

4.3.3 Recall for Baseline

We first measure the average dereferenceability of information in our sample. Let $\mathsf{data}(u, G)$ return the triples mentioning a URI u in a graph G, and, for a dereferenceable URI d, let $\mathsf{ddata}(d)$ denote $\mathsf{data}(d, \mathsf{deref}(d))$: triples dereferenceable through d mentioning d in some triple position. We define the *sample dereferencing recall* with respect to a sample graph G as:

$$\mathsf{sdr}(d, G) := \frac{\mathsf{ddata}(d)}{\mathsf{data}(d, G)}$$

Letting $G_\sim := \mathsf{merge}(\Gamma_\sim)$ denote the merge of our corpus, we measure $\mathsf{sdr}(d, G_\sim)$, which gives the ratio of dereferenceable triples for d mentioning d vs. unique triples mentioning d across the corpus. For comparability, we do not dereference d live, but use the HTTP-level information of the crawl to emulate $\mathsf{deref}(.)$ at the time of the crawl. We denote by $\mathsf{ddata}_\sim$ the average of $\mathsf{ddata}(d)$ for all $d \in D_\sim$, and by $\mathsf{sdr}_\sim$ the average of $\mathsf{sdr}(d, G_\sim)$ for all $d \in D_\sim$.

We also measure analogues of $\mathsf{ddata}_\sim$ and $\mathsf{sdr}_\sim$ where d must appear in specific triple positions: for example, if LTBQE dereferences a URI in the predicate position of a triple pattern, we are interested to know how often relevant triples – *i.e.*, triples with that URI in the predicate position – occur in the dereferenced document, how many, and what ratio when compared with the whole corpus.

Table 4.2 presents the results, where for different triples positions we present:

$$\begin{aligned}
|U_\sim| &: \text{number of URIs in that position,}\\
|D_\sim| &: \text{number of which are dereferenceable,}\\
\frac{|D_\sim|}{|U_\sim|} &: \text{ratio of dereferenceable URIs}\\
\mathsf{sdr}_\sim &: \text{as above, with std. deviation } (\sigma)\\
\mathsf{ddata}_\sim &: \text{as above, with std. deviation } (\sigma)
\end{aligned}$$

The row TYPE-OBJECT only considers the object position of triples with the predicate `rdf:type`, and the row OBJECT only considers object positions where the predicate is not `rdf:type`.

Position	$\lvert U_\sim \rvert$	$\lvert D_\sim \rvert$	$\frac{\lvert D_\sim \rvert}{\lvert U_\sim \rvert}$	$\mathbf{sdr}_\sim$		$\mathbf{ddata}_\sim$	
				avg.	σ	*avg.*	σ
ANY	1.87×10^7	8.37×10^6	0.449	0.51	± 0.5	17.26	± 97.15
SUBJECT	9.55×10^6	8.09×10^6	0.847	0.95	± 0.19	14.11	± 35.46
PREDICATE	4.77×10^4	745	0.016	7×10^{-5}	± 0.008	0.14	± 56.68
OBJECT	9.73×10^6	4.50×10^6	0.216	0.44	± 0.46	2.95	± 60.64
TYPE-OBJECT	2.13×10^5	2.11×10^4	0.099	0.002	± 0.05	0.07	± 29.13

Table 4.2: Dereferenceability results for different triple positions

The analysis provides some interesting practical insights into the LTBQE approach. Given a HTTP URI (without a common non-RDF/XML extension), we have a ~45% success ratio to receive RDF/XML content regardless of the triple position; for subjects, the percentage increases to ~85%, *etc.* If such a URI dereferences to RDF, we receive on average (at most) ~51% of all triples in which it appears across the whole corpus. Given a triple pattern with a URI in the subject position, the dereferenceable ratio increases to ~95%, such that LTBQE would work well for (possibly partly bound) query patterns with a URI in the subject position. For objects of non-type triples, the ratio drops to 44%. Further still, LTBQE would perform very poorly for triple patterns where it must rely on a URI in the predicate position or a class URI in an object position: the documents dereferenced from class and property terms rarely contain their respective extension, but instead often contain schema-level definitions. In summary, LTBQE performs well when URIs appear in the subject position of triple patterns, moderately when URIs appear in the object on a non-type triple, but poorly when URIs appear in the predicate or object of a type triple.

One may also note the high standard-deviation values in Table 4.2: these indicate that dereferenceability is often "all or nothing", particularly for object and predicate URIs. In Figure 4.2, for all 745 dereferenceable predicate URIs, we plot the distribution of the number of triples in the dereferenced document that contain the dereferenced term in the predicate position (log/log scale). Figure 4.3 shows the analogous distribution for dereferenceable object URIs in type triples. Although most such terms return little or no relevant information (*e.g.*, dereferencing the predicate in a triple pattern rarely yields triples where the dereferenced term appears as predicate), we see that a few predicates and values for `rdf:type` return a great many relevant triples in their dereferenced documents.[6] This might explains the high standard deviations: for example, although most predicates return no relevant information in their dereferenced document – and thus the probability of retrieving relevant information by dereferencing the predicate URI in a triple pattern is low – a few predicates dereference to large sets of relevant information.

4.3.4 Recall for Extensions

We now look at how the three LTBQE extensions can help to find additional data for generating query answers. Table 4.3 presents the average increase in raw information made available to LTBQE by considering `rdfs:seeAlso` and `owl:sameAs` links, as well as knowledge materialised through `owl:sameAs` and RDFS reasoning.

6 Many such examples for both classes and predicates come from the SUMO ontology: see, *e.g.*, `http://www.ontologyportal.org/SUMO.owl#subsumingRelation`

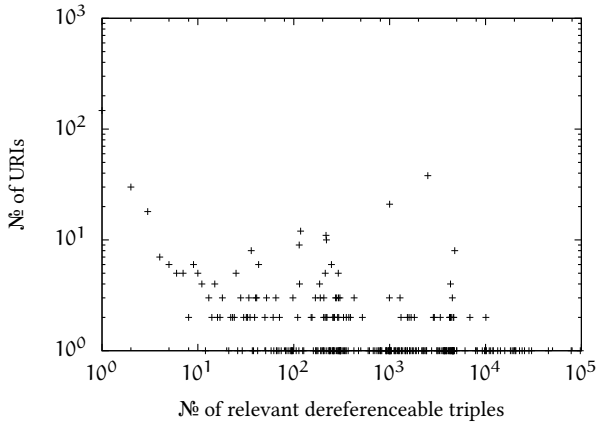


Figure 4.2: Relevant dereferenceable triple distribution for predicate URIs (log/log)

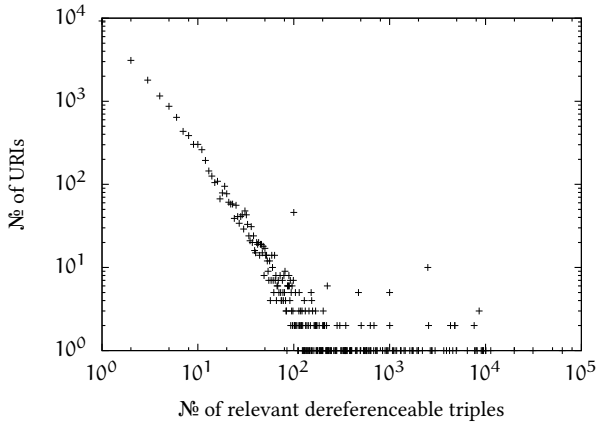


Figure 4.3: Relevant dereferenceable triple distribution for all type- object URIs (log/log)

$D_{\sim}^{+}$ indicates the subset of URIs in $D_{\sim}$ which have some relation to the extension, respectively: the URI has `rdfs:seeAlso` link(s), has `owl:sameAs` link(s), or has non-empty RDFS inferences. Also, $ddata_{\sim}^{+}$ indicates the analogous $ddata_{\sim}$ measure after the extension has been applied: this may involve data outside of the dereferenced document, such as documents reached through `rdfs:seeAlso` or `owl:sameAs` links, or static schema data.

4.3.4.1 *Benefit of Following `rdfs:seeAlso` Links*

We measured the percentage of dereferenceable URIs in $D_{\sim}$ which have at least one `rdfs:seeAlso` link in their dereferenced document to be ~2% for our sample (about 201 thousand URIs). Where such links exist, following them increases the amount of unique triples (involving the original URI) by a factor of 1.006× versus the unique triples in the dereferenced document alone. We conclude that, in the general case, considering `rdfs:seeAlso` information for the query processing will only marginally affect the recall increase of LTBQE.

Extension	$\lvert D^{+}_{\sim}\rvert$	$\frac{\lvert D^{+}_{\sim}\rvert}{\lvert D_{\sim}\rvert}$	$\frac{\mathsf{ddata}^{+}_{\sim}}{\mathsf{ddata}_{\sim}}$	
			avg.	σ
`rdfs:seeAlso` LINKS	2.01×10^5	0.02	1.006	±0.04
`owl:sameAs` LINKS & INFERENCE	1.35×10^6	0.16	2.5	±36.23
RDFS INFERENCE	6.79×10^6	0.84	1.8	±0.76

Table 4.3: Additional raw data made avaiable through LTBQE extensions

4.3.4.2 *Benefit of Following* `owl:sameAs` *Links & Including Inferences*

We measured the percentage of dereferenceable URIs in $D_{\sim}$ which have at least one `owl:sameAs` link in their dereferenced document to be ~16% for our sample. Where such links exist, following them and applying the EQ-* entailment rules over the resulting information increases the amount of unique triples (involving the original URI) by a factor of 2.5× vs. the unique (explicit) triples in the dereferenced document alone. The very high standard deviation of ±36.23 shown in Table 4.3 is explained by the plot in Figure 4.4 (log/log), which shows the distribution of the ratio of increase by considering `owl:sameAs` for individual URIs: we again see that although the plurality of URIs enjoy a small increase in raw data, a few URIs enjoy a very large increase. In more detail, Figure 4.5 gives a breakdown for URIs from individual domains, showing the number of URIs with an information increase above the indicated threshold due to `owl:sameAs`. The graph shows that, *e.g.*, some URIs from `nytimes.com` and `freebase.com` had an information increase of over 4000× (mostly due to DBpedia links); often the local descriptions were "stubs" with few triples.

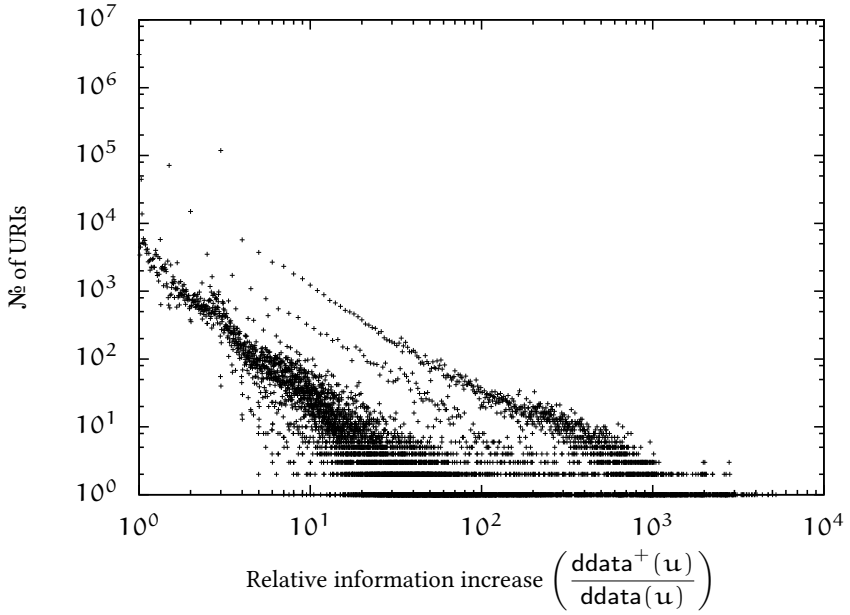


Figure 4.4: Distribution of relative information increases by materialising `owl:sameAs` information (log/log)

We conclude that, in the general case, `owl:sameAs` links are not so commonly found for dereferenceable URIs, but where available, following them and applying

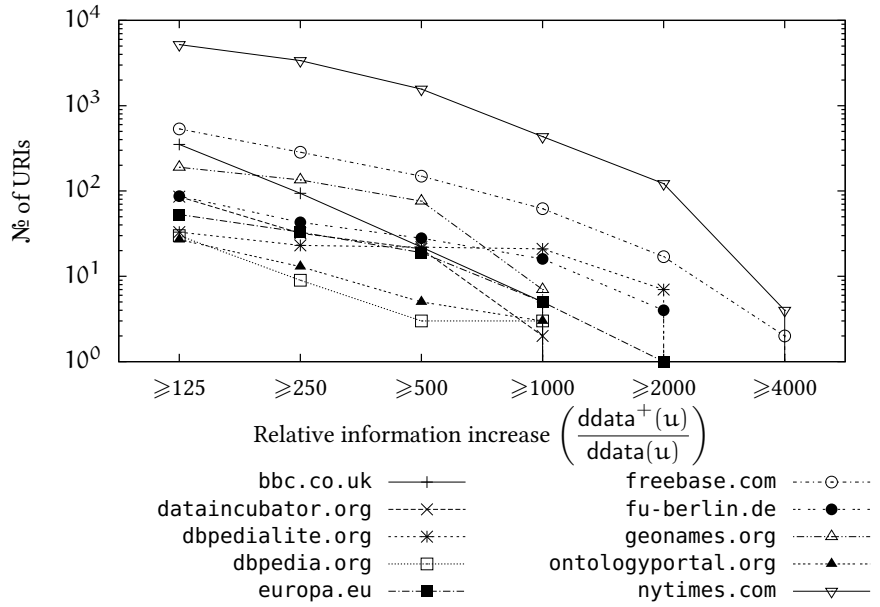


Figure 4.5: Binning of relative information increases by materialising `owl:sameAs` information per domain (log/log)

the entailment rules generates significantly more (occasionally orders of magnitude more) data for generating answers.

4.3.4.3 *Benefit of Including RDFS Reasoning*

With respect to our authoritative static schema data $\Psi_\sim$, we measured the percentage of dereferenceable URIs in $D_\sim$ whose dereferenced documents give non-empty entailments as ∼81%. Where such entailments are non-empty, they increase the amount of unique triples (involving the original URI) by a factor of 1.8× vs. the unique (explicit) triples in the dereferenced document. We conclude that such reasoning often increases the amount of raw data available for LTBQE query answering, and by a significant amount.

4.3.5 Discussion

Without looking at specific queries, in this section we find that, in the general case, LTBQE works best when a subject URI is provided in a query-pattern, works adequately when only (non-class) object URIs are provided, but works poorly when it must rely on property URIs bound to the predicate position or class URIs bound to the object position. Furthermore, we see that `rdfs:seeAlso` links, as proposed before [Hartig et al., 2009], are not so common (found in 2% of cases) and do not significantly extend the raw data made available to LTBQE for query-answering. Conversely, `owl:sameAs` links are a bit more common (found in 16% of cases) and can increase the available raw data significantly (2.5×). Furthermore, RDFS reasoning often (81% of the time) increases the amount of available raw data by a significant amount (1.8×).

As discussed previously, we use these results to modify our variant of LiDaQ which tries to minimise wasted remote lookups: aside from skipping URIs bound to non-join variables, this variant skips dereferencing predicate URIs bound in triple

patterns, or URIs bound to the objects of triples patterns where the predicate is bound to `rdf:type`, since we are unlikely to find data matching those patterns in the respectively dereferenced document (*cf.* Table 4.2). Of course, we still dereference these URIs for the purpose of dynamically collecting a set of schemata when performing RDFS reasoning.

These results show that we can expect a significant increase of query relevant data using our proposed novel reasoning extensions for LTBQE. We evaluate next to which extend these extension can also increase the query result and how feasible they are in practice.

4.4 QUERY BENCHMARKS

We wish to evaluate LiDaQ in a realistic, uncontrolled environment, answering SPARQL queries directly over a diverse set of Web of Data sources. To guide this evaluation, we first survey existing Linked Data SPARQL benchmarks and look at how other relevant systems evaluate their approaches (Section 4.4.1). We conclude that no benchmark offers a large and diverse range of suitable SPARQL queries for LTBQE, and we thus propose *QWalk*: a novel benchmark methodology tailored for testing LTBQE-style query-answering approaches over the broader Web of Data (Section 4.4.2). Final evaluation setup and results are presented later in Section 4.5.

4.4.1 Existing Linked Data SPARQL Benchmarks

We now look at existing SPARQL benchmark frameworks and how they have been used to evaluate various "live" Linked Data query processing approaches in the literature. The main purpose of this survey is to study the query types and the benchmark environments used in order to inform our own evaluation of LTBQE and its extensions. We find that various published SPARQL benchmark frameworks focus on Linked Data query processing and some of the provided queries could be re-used, but that existing papers evaluate their approaches with respect to either hand crafted queries or domain-specific queries, whereby all are restricted to one or few domains of data.

4.4.1.1 *Benchmark Frameworks*

We now discuss existing Linked Data SPARQL benchmarking frameworks. Since we aim to run our evaluation over Linked Data sources *in situ* – and to demonstrate the real world behaviour of the methods discussed herein – we focus on query frameworks designed to run over real-world Linked Data, where we do not treat SPARQL benchmarks designed to run over synthetic datasets such as *LUBM* [Guo et al., 2005], *BSBM* [Bizer and Schultz, 2009], or *SP²Bench* [Schmidt et al., 2008a,b]. The problem with the synthetic benchmark is, that the generated queries cannot be run directly over the Web of Data and one would need to setup and simulating a controlled "Web" environment. In fact, to the best of our knowledge, only two relevant frameworks have been proposed and are suitable for benchmarking LTQBE:

FEDBENCH – The FedBench [Schmidt et al., 2011] framework is designed specifically for testing Linked Data querying scenarios. Queries are formulated against three data collections [Schmidt et al., 2011]:

1. a Life Science Data Collection, which includes datasets like KEGG, ChEBI, DrugBank and DBPedia;

2. a synthetic dataset from the SP^{2}Bench framework [Schmidt et al., 2008a,b]; and

3. a general Linked Open Data Collection, which includes datasets like DBpedia, GeoNames, Jamendo, LinkedMDB, The New York Times and Semantic Web Dog Food.

Three independent query sets are then defined. The first query set focuses on features of particular interest for federated query engines, such as the number of involved sources, (interim) query results size (e.g join complexities), and so forth. The second query set consists of the original SP^{2}Bench queries. The third query set provides 11 Linked Data specific queries[7], which consist of SPARQL Basic Graph Patterns in the form of 6 path queries, 3 star queries and 2 mixed/hybrid queries. This third set is thus of particular interest to us.

DBSPB – The DBpedia SPARQL Benchmark [Morsey et al., 2011] contains a set of SPARQL queries distilled from real-world DBpedia logs, consisting of 31.5 million queries issued by various users and agents over a four-month time-frame in 2010. The raw set of queries is reduced to a total of 35 thousand queries after less frequently occuring query shapes were removed. These 35 thousand queries are clustered to generate 25 templates which characterise the larger set. These templates can be "instantiated" to create new queries from DBpedia data. The templates consist of Basic Graph Pattern queries with 1–5 triple patterns, but also contains templates that generate queries with various combinations of SPARQL query features (*e.g.*, `OPTIONAL`–`FILTER`–`DISTINCT`, `UNION`–`FILTER`, and so forth).

4.4.1.2 *Evaluation of Linked Data Query Approaches*

We now look at the specifics of how various authors have evaluated their proposed approaches for querying Linked Data. We focus on the evaluation of non-materialised engines: *i.e.*, we focus on query proposals that (typically) involve accessing remote data at runtime. In particular, we summarise which query sets were used, what data were used, and how the benchmark was setup.

[LTBQE1] In the work which initially proposed the explorative LTBQE model, Hartig et al. [2009] used four manually crafted queries to demonstrate the feasibility of the approach in the real-world. The results present query time, number of results and number of sources involved. In addition, the authors used 12 BSBM queries (for synthetic data) and a controlled setup with a proxy server to evaluate the query time for different fetch-scheduling strategies.

[LTBQE2] This work studies how different in-memory data structures might influence the performance of the original LTBQE approach [Hartig and Huber, 2011]. The proposed data structures are evaluated with different datasets and the query performance was again benchmarked with BSBM.

[LTBQE3] Another proposed LTBQE extension is to cache query relevant data to improve the result completeness of the LTBQE approach [Hartig, 2011a]. The published evaluation uses a real-world use-case, called the FOAF Letter application[8], which involved five query templates – with a mix of different shapes and SPARQL features – instantiated for 23 people, giving a total of 115 queries[9].

7 `http://code.google.com/p/fbench/wiki/Queries#Linked_Data_(LD)`

8 The application provides a service for users with FOAF profiles to keep track of their social network by periodically checking updates and suggesting new connections using the LTBQE approach. See `http://linkeddata.informatik.hu-berlin.de/foafletter/`

9 `http://squin.sourceforge.net/experiments/CachingLDOW2011/`

Reference	Queries			Measures			Live	Evaluation Setup
	count	*type*	*published*	*time*	*results*	*sources*		
LTBQE1 a)	4	Custom	X	✓	✓	✓	✓	Single run
LTBQE1 b)	12	Custom	X	✓	X	X	X	BSBM query mixes, RAP Pubby setup
LTBQE2	200	Custom	✓	✓	X	✓	X	BSBM query mixes, RAP Pubby setup
LTBQE3	115	Custom	✓	✓	✓	✓	✓	3 runs per query
LTBQE4	3	Custom	✓	✓	✓	✓	✓	6 runs per query (1^{st} run warmup)
LT10	8	Custom	✓	✓	✓	✓	X	Controlled with 2 second delay proxy
SIHJoin	10	Custom	✓	✓	X	X	X	CumulusRDF Linked Data proxy
FedX	11	FedBench	✓	✓	X	X	X	Local copies of SPARQL endpoints
LH10	390	Custom	X	✓	✓	✓	X	Simulated HTTP lookups
SPLENDID	14	FedBench	✓	✓	X	✓	X	Local replication of SPARQL endpoints
SPARQL-DQP	7	Custom	✓	✓	X	X	X	Amazon EC2 instance

Table 4.4: Summary about number of queries, evaluation measures and setup in the literature about live Linked Data query methods.

[LTBQE4] In follow up work, Hartig [2011b] discusses and evaluates different strategies for the query execution order. The impact of the query evaluation order is evaluated by executing 3 manually crafted real-world queries six times live over the Web of Data. The three queries are a mix of star and path-shaped Basic Graph Patterns with 6–8 elements.

[LT10] Ladwig and Tran [2010] investigate three different strategies to execute SPARQL queries over the Web of Data and systematically analyse and compare them. The benchmark contains 8 queries and was executed in a controlled environment with a local proxy server.

[SIHJOIN] Ladwig and Tran [2011] present experiments on actual real-world datasets and on synthetic data to evaluate the benefit of their proposed symmetric hash-join operator against the non-blocking iterator proposed by Hartig et al. [2009]. Their evaluation is executed in a controlled environment (hosted using CumulusRDF [Ladwig and Harth, 2011]) and uses 10 manually created queries (similar to the FedBench Linked Data query set).

[FEDX] Schwarte et al. [2011] evaluate their proposed query engine for federating SPARQL endpoints using the (entire) FedBench SPARQL benchmark framework.

[LH10] Li and Heflin [2010] investigated using reformulation trees to organise local summarised knowledge about sources; they also investigate how OWL reasoning can be used for the explorative query execution. The evaluation uses a synthetic data set and manually selected subsets of the Billion Triple Challenge 2010 dataset. Queries are created manually. The experiment was executed in a controlled environment and HTTP lookups were simulated.

[SPLENDID] Görlitz and Staab [2011] benchmark their SPARQL federation approach (based on VoID descriptions) using the life science and cross domain query sets from FedBench. The benchmark was conducted over a local replication of the datasets and endpoints.

[SPARQL-DQP] Aranda et al. [2011] evaluate their federated SPARQL techniques using modified version of the Life Science queries in the FedBench framework, where additional SPARQL features are added. The queries are then run over four endpoints, two of which are replicated locally.

To provide a good overview – and for ease of comparison – we summarise this survey of the evaluation of Linked Data query systems in Table 4.4.

4.4.1.3 *Discussion*

We see that there is a lot of diversity in how different Linked Data query proposals have been evaluated. First and foremost, we highlight that few benchmarks have been proposed for real-world Linked Data; perhaps the most agreed upon is FedBench. Furthermore, most benchmarks and evaluations involve either a handful of manually crafted queries designed to run over a small number of sources (FedBench), or involve a larger number of (semi-)automatically generated queries but are tied to a specific domain (*e.g.*, DBPSB) or schema of data (*e.g.*, FOAF Letter). Furthermore, we note that few engines run their queries live over remote resources, but rather prefer to replicate raw content or endpoints within a controlled environment. In summary, we find that no live Linked Data query engine has been evaluated in an uncontrolled, real-world setting for a large set of diverse queries: the closest such evaluation is probably FOAF Letter [Hartig, 2011a], which was run

live, but which involved a handful of resources and was centred specifically around FOAF profiles.

4.4.2 QWalk: Random Walk Query Generation

Given the shortcomings of existing benchmarks, we propose a new benchmark framework – called *QWalk* – that is tailored to link traversal based approaches and builds a large set of queries that are answerable over a diverse set of real-world sources that use different schemata. The core idea is to take a large crawl of the Web of Data (in this case, the BTC'11 dataset) and to conduct random walks of different shapes and lengths through the corpus to generate Basic Graph Patterns. The walk is guided to ensure that it crosses documents through dereferenceable links, such that it should return results through an LTBQE-style approach.

4.4.2.1 *Query Shapes*

To inform the types of queries we generate, we take observations from the work of Arias et al. [2011], who analyse the SPARQL queries logs of the DBPedia and Semantic Web Dog Food (SWDF) servers. They found that most queries contain a single triple pattern (66.41% in DBPedia, 97.25% in SWDF). The maximum number of patterns found was 15, but such complex queries occurred only rarely. The most common forms of joins involved subject–subject (59–61%), subject–object (32–36%) and object–object (4–5%); few joins involving predicate variables were found in general. As such, most queries with multiple patterns are star-shaped, with a few path shaped queries. Star-shaped joins typically had a low "fan-out", where 27% of the DBpedia queries had a fan-out of three, and 3.7% had a fan-out of two; the bulk of the remaining queries were single-pattern with a trivial fan-out of one, but went up to a maximum of nine. The lengths of paths in the query were mostly one (98%) or two (1.8%); very few longer paths were found.

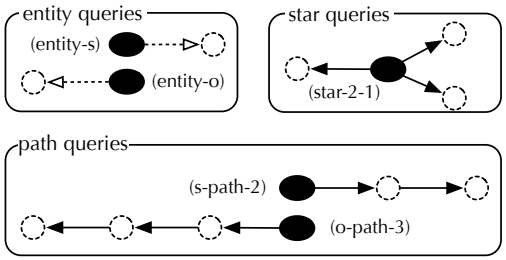


Figure 4.6: Visualisation of some example benchmark queries; dotted lines represent query variables

Along similar lines, for our benchmark we generate queries of elemental graph shapes as depicted in Figure 4.6, *viz.*, entity, star and path queries. We now describe these query types in more detail.

ENTITY QUERIES (ENTITY-<S|O|SO>)

ask for all available triples for an entity. We generate three types of entity queries, asking for triples where a URI appears as the subject (ENTITY-S); as the object (ENTITY-O); as the subject *and* object (ENTITY-SO). These types of queries are very common in Linked Data Browsers, interfaces, or for dynamically serving dereferenceable Linked Data content. An example for ENTITY-SO would be displayed in Query 4.11

```
SELECT DISTINCT ?p1 ?o ?s ?p2
WHERE {
  <d> ?p1 ?o .
  ?s ?p2 <d> .
}
```

Query 4.11: Example for an ENTITY-SO query

STAR QUERIES (STAR-<S3|O3|S1-O1|S2-O1|S1-O2>)

always have three acyclic triple patterns which share exactly one URI (called the centre node) and where predicate terms are constant. We do not consider stars involving predicate variables since (as discussed) they are rarely used in practice [Arias et al., 2011]. We generate four different variations of such queries, differing in the number of triple patterns in which the centre node appears at the subject (S) or object (O). Thus, each query has 4 constants and 3 variables. An example for STAR-S2-O1 is be Query 4.12.

```
SELECT DISTINCT ?o1 ?o2 ?s1
WHERE {
  <d> foaf:knows ?o1 .
  <d> foaf:name ?o2 .
  ?s1 dc:creator <d> .
}
```

Query 4.12: Example for a STAR-S2-O1 query

PATH QUERIES (<S|O>-PATH-<2|3>)

have 2 or 3 triple patterns which form a path such that precisely two triple pattern share the same variable. Exactly one triple pattern has a URI at either the subject or object position and all predicate terms are constant. As before, we do not consider paths connected through predicate variables since they are rare [Arias et al., 2011]. We generate four different sub-types: path shaped queries of length 2 and 3 in which either the subject or object term of one of the triple patterns is a constant. An example for S-PATH-2 is the following:

```
SELECT DISTINCT ?o1 ?o2
WHERE {
  dblpP:HartigBF09 foaf:maker ?o1 .
  ?o1 foaf:name ?o2 .
}
```

Query 4.13: Example for a S-PATH-2 query

QUERY GENERATION

We generate queries from the aforementioned BTC'11 dataset. In total, we generate 100 `SELECT DISTINCT` queries for *each* of the above 11 query shapes using random walks in our corpus. To help ensure that queries return non-empty results (in case there are no HTTP connection errors or time outs) we consider dereferenceable information and generate queries as follows:

1. We randomly pick a pay-level-domain available in the set of confirmed dereferenceable URIs $D_{\sim}$.
2. We then randomly select a URI from $D_{\sim}$ for that pay-level-domain.
3. We generate appropriate triple patterns from the dereferenceable document of the selected URI based on the query shape being generated.

- If path-shaped queries are being generated, the URI for the next triple pattern is selected from the dereferenceable URIs connected to the previous URI, as per a random walk.

4. One variable is randomly chosen as distinguished (returned in the `SELECT` clause) and other variables are made distinguished with a probability of 0.5.

By randomly selecting a pay-level-domain first (as opposed to randomly selecting a URI directly), we achieve a greater spread of URIs across different datasets. The result of the QWalk process is a large set of diverse queries with different elemental shapes that – according to the sampled data – should be answerable through LTBQE methods over real-world data in a realistic scenario (accessing remote sources).

4.5 EVALUATION

4.5.1 Setup

When running queries directly over remote sources, various challenges come to the fore, including slow HTTP lookups, unpredictable remote server behaviour, high fan-out of links to traverse, the need for polite access (in terms of delays between lookups and respecting `robots.txt` policies), and so forth. As we have seen, most works have evaluated LTBQE-style approaches in controlled environments – using proxies and simulated remote access – or only for a small number of queries or sources in uncontrolled environments. Conversely, we now wish to stress-test LTBQE – and our proposed reasoning extensions – for a large range of diverse queries in uncontrolled environments, and to characterise how these methods handle the aforementioned challenges.

Based on the discussion of the previous section, we select three complementary benchmarks to run with LiDaQ:

1. FedBench Linked Data Queries, which offers a few manually crafted queries designed to run over a number of different domains;
2. DBpedia SPARQL Benchmark (DBPSB), which offers the potential to generate many queries designed to run over one domain, and are based on real-world query logs;
3. QWalk, which offers a large selection of queries that can be run directly over a diverse set of sources.

We execute all queries directly over the Web of Data in an uncontrolled environment without any replication, proxies or simulation of HTTP lookups. Thus, the measured values reflect the expected query behaviour in a real application scenario and allows us to discuss the feasibility of LiDaQ – and of LTBQE query approaches in general – within such a setting.

We first discuss the measures that we take and the environment in which the experiments are run (Section 4.5.1.1). We then introduce the configurations of LiDaQ that we evaluate (Section 4.5.1.2). Then we present the results of the FedBench run (Section 4.5.2.1), the DBPSB run (Section 4.5.2.2), and the QWalk run (Section 4.5.2.3).

4.5.1.1 *Environment and Measures Taken*

All evaluation is run on one server with 4 GB of main memory. To ensure polite behaviour, we enforce a per-domain (specifically per-PLD) minimum delay of 500

ms between two sequential HTTP lookups on one domain. Furthermore, we use a (generous) query timeout of 2 hours.

Given that we run queries live over a diverse set of remote sources, we often encounter some "non-deterministic" behaviour: in particular, a source may be readily accessible during some query runs, but may be unresponsive or not return at all in others. In other words, different HTTP-level issues can occur at different times for the same source. During initial experiments, we thus encountered that result size and execution time can differ for the same query and setup between several benchmark runs. This "inconsistent" query behaviour is explained by the fact that we encountered various HTTP-level issues between different executions for the same query and setup.

We thus define the straightforward notion of a *benchmark stable query*, which we consider in our results to help with comparability across different setups. We assume that a query has a core set of relevant sources, that is, the URIs which are accessed in all different setup configurations (introduced in the following section). A *benchmark stable query* is a query for which the response codes for each of the core URIs is the same across all setups runs.

For each query in each evaluation framework, we record the following six measures:

RESULT:	the number of distinct results,
TIME:	the total time taken to execute the query,
FIRST:	time elapsed until the first result was returned,
LOOKUPS:	total number of HTTP GET lookups,
DATA:	total number of raw triples retrieved, and
INFERRED:	total number of unique triples inferred.

In the following, we concentrate our discussion around the RESULT and TIME measures since our focus is to study to which extend our reasoning extensions can improve the result recall and at the same time how the query performance is influenced

These measures together characterise the behaviour of the LIDAQ engine while processing queries under various setups. However, raw counts of query results can sometimes exhibit outliers due to cross-products inherently caused by joins. We illustrate this with an example QWalk query, which caused some surprising behaviour when RDFS reasoning was used.

Example 4.10. For QWalk, we encountered the following Query 4.14:

```
SELECT DISTINCT ?s0 ?o0 ?s1
WHERE {
    ebiz: owl:imports ?o0 .
    ?s0 rdfs:seeAlso ebiz: .
    ?s1 rdfs:isDefinedBy ebiz: .
}
```

Query 4.14: Example of evaluation query

Without reasoning, upon dereferencing the `ebiz:` URI, we found 1 binding for `?o0`, 2 bindings for `?s0` and 199 bindings for `?s1`, yielding a total of $1 \times 2 \times 199 = 398$ results.

However, with RDFS reasoning enabled, the schema document for RDFS (the so-called "`rdfs.rdfs`" document) authoritatively defines the property

`rdfs:isDefinedBy` to be a sub-property of `rdfs:seeAlso`. Thus, the 199 bindings for `?s1` are added to `?s0`, yielding 201 bindings, and a total of $1 \times 201 \times 199 = 39,999$ results, giving a two orders of magnitude increase.

The query results for the above example involves a lot of repetitions of terms: each term bound to `?s0` or `?s1` would appear about 200 times. Hence we add an additional measure to help loosely characterise the "redundancy-free content" of the results: we summate the number of unique result **terms** found for each distinguished variable in the results. In the above example, this would give $1 + 2 + 199 = 202$ terms without reasoning, and $1 + 201 + 199 = 39,999$ terms with reasoning.

4.5.1.2 *LiDaQ Configurations Evaluated*

For LiDaQ, given the various extensions, various ways of collecting RDFS data, and the option to turn off/on our reduction of sources, we have thirty-two combinations of possible setups, where we choose to evaluate the impact of each extension on its own. In addition, we benchmark the result recall and query time if all extensions are enabled. Therefore, we have the following ten configurations:

CORE (CORE): We dereference all URIs appearing in the query and during the query execution, independent from their triple position or role in joins. No extensions are included. This setup serves primarily as reference to the basic LTBQE profile.

REDUCED CORE (CORE^{-}): Our reduced configuration uses our optimised source selection to decide which URIs are query relevant and should be dereferenced. We do not dereference URIs bound to non-join variables, or (unless dynamically retrieving schemata) URIs bound to predicates or values for `rdf:type`. We expect in theory the smallest amount of results and also the fastest query time. The following extensions are built upon this reduced profile, not CORE.

WITH `rdfs:seealso` LINKS (SEEALSO): This setup extends the CORE^{-} setup by following `rdfs:seeAlso` links. Based on our empirical analysis, we expect that only a small number of queries will be affected given that ~2% of the URIs contain these information in the dereferenceable document and that they make little additional raw data available. As such, we expect that this extension occasionally increases the query time without affecting the result recall.

WITH `owl:sameas` LINKS AND REASONING (SAMEAS):

This configuration extends CORE^{-} by considering `owl:sameAs` links and inference. We expect for all queries an increase in returned results and the number of lookups. Based on our empirical study, this should affect a moderate number of queries given that such information is available for ~16% of resources, where, in such cases, inference makes on average 2.5× more raw data available.

WITH RDFS INFERENCE ($\text{RDFS}_{[s|d|e]}$): this configuration extends the CORE^{-} setup by performing inferences for the RDFS ruleset relying on retrieved schema information. Based on static schema data, our empirical analysis suggested that RDFS reasoning affects ~84% of resources for which it makes about 1.8× more raw data available. However, we investigate three sub-configurations based on the methods described in Section 4.2.1.3:

STATIC (RDFS_{s}): uses the *static schema* which we extracted earlier from the BTC'11 dataset;

DIRECT (RDFS_d): collects *direct schemata* by dynamically dereferencing the predicates and values for `rdf:type`;

EXTENDED (RDFS_e): collects *extended schemata* by dynamically following recursive links from the direct schemata.

COMBINED ($\text{COMB}_{[\text{s}|\text{d}|e]}$): this benchmark setup combines all extensions for the previously mentioned configurations of schema collection. With this configuration, we expect the highest number of results, query time and processed and inferred statements.

For ease of reference, Table 4.5 gives an overview of these test configurations, and which features are enabled or disabled.

Label	*reduced sources*	`rdfs:seeAlso`	`owl:sameAs`	*static RDFS*	*direct RDFS*	*extended RDFS*
CORE						
CORE^-	✓					
SEEALSO	✓	✓				
SAMEAS	✓		✓			
RDFS_s	✓			✓		
RDFS_d	✓				✓	
RDFS_e	✓					✓
COMB_s	✓	✓	✓	✓		
COMB_d	✓	✓	✓		✓	
COMB_e	✓	✓	✓			✓

Table 4.5: Overview of the ten LIDAQ benchmark configurations

4.5.2 Results

4.5.2.1 *FedBench Results*

Our first experiment uses the FedBench Linked Data Queries (Section 4.4.1.1) to measure the potential benefit of our proposed extensions and optimisations. These 11 cross-domain queries (denoted LD1–LD11) are designed to return a non-empty result set if executed over the Web with an LTBQE-style approach. The queries (along with results and discussion) can be found in Appendix C. Our first observation is that 4 out of the 11 manually crafted FedBench queries contain explicit `owl:sameAs` query patterns. This fact ties back with our initial motivation that including `owl:sameAs` information is important to answer queries across diverse sources. In the case of FedBench, these `owl:sameAs` relations are included explicitly; for our extension this would not be necessary (though we leave them in for comparability across LIDAQ configurations that include/exclude `owl:sameAs` inference).

QUERY TESTING In initial tests, we only received results for 6 out of the 11 FedBench queries. A manual inspection of the queries and their relevant data revealed

that the empty results were either caused by (i) access forbidden by `robots.txt` or (ii) updates in the DBpedia datasets (which, in itself, lends strength to the arguments for live querying approaches).

In the first category, there was only 1 query:

DISALLOWED BY `robots.txt` (LD7):

> This particular query requires data from the `geonames.org` domain, especially from the subdomain `sws.geoname.org`. The query fails because the access for resources on this subdomain is disallowed via the `robots.txt` protocol.[10]

We do not wish to contravene the Robots Exclusion Protocol and so we do not run this query in the main evaluation.

In the second category – those caused by being out-of-sync with DBpedia – there were 4 queries, which we fixed manually:

MISSING LANGUAGE TAG (LD9):

> One query was missing a language tag. We changed the query literal `"Luiz Felipe Scolari"` by adding the correct English language tag: `"Luiz Felipe Scolari"@en`.

OUTDATED PREDICATES (LD8 – LD10):

> Three queries contain the predicate `skos:subject`, which was initially proposed for SKOS and used by DBpedia. However, the SKOS Working Group later chose to instead reuse `dcterms:subject` and DBpedia followed suit. We thus changed the all query predicates from `skos:subject` to `dcterms:subject` where applicable.

Thus, we keep 10 of the 11 original queries, where 4 are adapted slightly to work against current versions of DBpedia, and where 1 is dropped due to `robots.txt` issues. In addition, since we count results, we add to all queries the `DISTINCT` solution modifier to eliminate duplicates. Our updated queries are online[11]. However, we still find that some of the updated queries do not return results: LD6 and LD9 return no results for any configuration (or any run), and LD10 only sometimes returns results when `owl:sameAs` is considered; these are due to mismatches between the given queries and remote data that we could not easily fix without dramatically changing the original query (see Appendix Appendix C for details).

OVERVIEW OF EXPERIMENTS In total, using LIDAQ, for each configuration, we executed each FedBench query live over the Web once a week for four weeks. In Appendix Appendix C, for each individual query, we provide detailed results and discussion (selecting the best of the four runs). We also include per-query comparison to the existing SQUIN library for LTBQE [Hartig et al., 2009], which we generally find to be considerably faster, but which, to the best of our knowledge, does not include politeness policies; we thus exclude it for later larger-scale experiments.[12] Herein, we focus on the general impact of our extensions on the repeatability and reliability of the results across the four runs, averaged across all queries.

DETAILED RESULTS Given the number of queries (10), configurations (11) and measures (6), we rather present detailed discussion of the FedBench results for each individual query in Appendix C.

10 See `http://sws.geonames.org/robots.txt`, which states that all agents are disallowed.
11 `http://code.google.com/p/lidaq/source/browse/queries/fedbench.txt`
12 We found queries for which it sends at least eight lookups per second to the same server.

Summarising herein, we observe that LTBQE works well for some simpler FedBench queries, but struggles in an uncontrolled environment for complex queries that require accessing a lot of sources. For example, even in the baseline CORE$^-$ configuration, LD11 required performing 1,125 lookups, and the most complex configuration – COMB$_e$ – attempted 17,996 lookups. Relatedly, we generally observe that LTBQE extensions perform well for simple queries, but exacerbate performance issues for complex queries. In fact, although RDFS and `owl:sameAs` extensions work well on domains like `data.semanticweb.org`, configurations that involve following same-as or schema links struggle for data-providers such as DBpedia, which offer many such links (both internal and external). On a more positive note, we find that CORE$^-$ often offers significant time savings over CORE with minimal effect on result sizes.

In addition, results sizes can become quite large (*e.g.*, LD11 returns 196,448 results in one configuration): the given SPARQL queries often contain numerous result variables in the `SELECT` clause (*e.g.*, 5 for LD11) and results require a product for variables with compatible mappings [Pérez et al., 2009]. For example, DBpedia often contains a large number of labels for resources in different languages, where asking for the labels of result resources may multiply the raw result sizes by a factor of ten or more; as discussed previously, such behaviour has a cumulative effect.

In general, we would also expect that the results given by CORE$^-$ should be fewer or equal than for all other configurations, which are monotonic extensions; similarly, we would expect equal or more results in COMB$_x$ than RDFS$_x$, SAMEAS and SEEALSO (where x is one of the schema configurations). This expectation held true in practice for a number of the earlier queries (LD1–3), where for queries LD1 and LD3, the various extensions, including reasoning, found many more results. This monotonic increase also held true for certain other cases (*e.g.*, with the exception of COMB$_e$, LD4 shows this behaviour). However, it did not hold true in later queries: even selecting the best run from a span of four weeks, the unrepeatability of results played a major role in this evaluation. We thus now focus on characterising this issue.

RELIABILITY RESULTS Running complex queries live over networks of remote sources raises the question of reliability and repeatability. We now focus on how the results varied across the four runs to get a better idea of the repeatability of LTBQE/LIDAQ in a realistic setting. We summarise the average number of results and the corresponding standard deviation for each configuration and query across all four runs in Table 4.6 and Table 4.7.

The query LD11 in particular shows some unreliable behaviour across the four runs, where we estimated the absolute deviations to be between ~28–140% of the mean, depending on the LIDAQ configuration: as aforementioned, this query required between 1,103–17,996 lookups. With this exception aside, across all other queries, the CORE, CORE$^-$ and SEEALSO configurations access the fewest sources and produce reliable results across the four runs. The results for the other variations – which include reasoning extensions – are less reliable in general. LD5 and LD10 show high deviations in the number of results returned for SAMEAS, LD5 shows high deviations for dynamic schema configurations, and LD4 shows high deviations for configurations involving RDFS reasoning (though not for combined configurations). In terms of absolute deviation as a percentage of the mean, we computed that the results for the other setups vary somewhere between ~2–11% for most of the queries.

CONCLUSIONS In summary, when running the queries live over remote sources, we see complex and unpredictable behaviour across different configurations and

Setup	LD1		LD2		LD3		LD4		LD5	
	avg.	σ	*avg.*	σ	*avg.*	σ	*avg.*	σ	*avg.*	σ
CORE	333	±0	185	±0	191	±0	50	±0	21.5	±24.83
CORE^{-}	333	±0	185	±0	190.75	±0.5	50	±0	24	±22.32
SEEALSO	333	±0	185	±0	190.75	±0.5	50	±0	21.5	±24.83
SAMEAS	527.25	±3.95	185	±0	908.25	±35.53	146	±0	67.75	±135.5
RDFS$_s$	380	±0	185	±0	246	±0	50	±0	17.5	±21.24
RDFS$_d$	380	±0	185	±0	246	±0	50	±0	20.25	±23.41
RDFS$_e$	380	±0	185	±0	246	±0	37.5	±25	8	±9.38
COMB$_s$	674.5	±14.15	185	±0	1,385	±161.85	137	±92.57	0	±0
COMB$_d$	662.5	±14.71	185	±0	1,428	±122.36	151.25	±100.85	3.5	±7
COMB$_e$	674.5	±14.15	185	±0	1,428	±122.36	88.5	±59.29	—	—

Table 4.6: Average result size and standard deviation for four query runs for LD1–LD5

Setup	LD6		LD8		LD9		LD10		LD11	
	avg.	σ	*avg.*	σ	*avg.*	σ	*avg.*	σ	*avg.*	σ
CORE	0	±0	9.5	±10.97	0	±0	0	±0	21,801.75	±6,255.72
CORE^{-}	0	±0	9.5	±10.97	0	±0	0	±0	14,322	±12,237
SEEALSO	0	±0	19	±0	0	±0	0	±0	19,096	±9,373.88
SAMEAS	0	±0	10,535.5	±12,165.35	0	±0	2,512.5	±2,973.81	8,466	±10,940.75
RDFS$_s$	0	±0	9.5	±10.97	0	±0	0	±0	1,745.5	±3,491
RDFS$_d$	0	±0	1	±2	0	±0	0	±0	2,757.25	±3,508.19
RDFS$_e$	0	±0	3	±6	0	±0	–	–	–	–
COMB$_s$	0	±0	14,096.75	±16,295.82	0	±0	812.25	±1,624.5	71,438	±21,634.8
COMB$_d$	0	±0	–	–	0	±0	0	±0	177,596.5	±61,929.13
COMB$_e$	0	±0	–	–	–	–	–	–	50,660	±71,289.96

Table 4.7: Average result size and standard deviation for four query runs for LD6–LD11

across time: remote sources may give different responses at different times (*e.g.*, may give `50x` errors during high server load), and the failure of an important source may break traversal at that point. We also see that reasoning extensions, particular those involving dynamic schema collection, often make the query behaviour more unreliable by trying to access more sources.

The FedBench queries predominantly request data from a few central sites: the first four queries (LD1–4) are based around the `data.semanticweb.org` data provider, and provide generally stable results. Other queries also rely on the hosts `www4.wiwiss.fu-berlin.de` and `dbpedia.org` and generally demonstrate less stable behaviour. Taking the former domain, for example, `owl:sameAs` extensions can cause erratic behaviour, potentially due to errors in how the relation is used for the DailyMed and LinkedCT datasets.[13] For DBpedia, the schema descriptions of class and property terms are hosted in individual documents and often interlink with related sources like Yago [Suchanek et al., 2008] and CYC, causing to many documents being requested when schemata are dynamically retrieved, leading to unstable behaviour for such configurations with respect to these queries. Given that the queries are restricted to a few data providers, politeness policies play a crucial role: the amount of time-delay enforced between subsequent lookups to the same host can be a major factor of performance.

In general, from the 11 original FedBench queries, which were designed to be run using LTBQE-style approaches, 4 queries show promising results, 3 return no results (2 involve access disallowed by `robots.txt`), and the remaining 4 queries show unpredictable behaviour across different runs and configurations. Some of the more complex queries involve accessing thousands and tens of thousands of sources at runtime. By requesting even more sources, our proposed reasoning extensions can aggravate reliability issues. This calls into question the practicality of the LTBQE approach (and our reasoning extensions) in uncontrolled environments for complex queries that span multiple sites and require many sources to answer. We will later show that LTBQE works well for simple queries with our QWalk generated queries.

4.5.2.2 *DBPSB Results*

The FedBench Linked Data queries are designed specifically for evaluation using LTBQE-style engines such as LiDaQ. We now rather look at the DBpedia SPARQL benchmark [Morsey et al., 2011] (DBPSB; Section 4.4.1.1): a generic Linked Data SPARQL benchmark containing realistic queries based on popular patterns mined from real-world DBpedia access logs. Thus, DBPSB should give us an indication as to how well LTBQE can cope with (non-tailored) queries that are based on those frequently run by users against the materialised DBpedia SPARQL engine. As we will see, LTBQE and its extensions stuggle for this query suite.

QUERY TESTING In total, the DBPSB query set consist of 25 different query templates (denoted DB1–25).[14] Each template has an associated template query that can be run against a DBpedia SPARQL endpoint to instantiate a set of concrete queries used for evaluation: each template query has a subset of *template variables* that are bound to create a new evaluation query (one query per binding) in this manner, where the rest of the variables are left as is for the evaluation query.

13 We observed and reported such problems before: see `https://groups.google.com/forum/?fromgroups#!topic/pedantic-web/rXQPcFLMOi0` for detailed discussion.

14 Originally taken from `http://dbpedia.aksw.org/benchmark.dbpedia.org/Queries2011.txt`. Formatted and annotated templates can be found at `http://code.google.com/p/lidaq/source/browse/queries/dbpsb.txt`

In a first step, we manually inspected the DBPSB query templates and ruled out those which LTBQE would clearly not be able to answer. We thus initially ruled out 16 queries:

UNSUPPORTED `optional` FEATURE (8 queries):

> A total of 8 query templates use the `OPTIONAL` keyword, which can only be correctly evaluated over a closed dataset since it is a non-monotonic feature [Hartig, 2012]. Problematically, if data are not available to match the `OPTIONAL` clause, SPARQL specifies that `UNBOUND` should be returned. Returning `UNBOUND` is a definitive answer that the data are not available, and can be tested elsewhere in a `FILTER` clause, allowing features such as negation-as-failure. LTBQE cannot definitively say that data are not available (unless a bounded dataset is considered [Hartig, 2012]).

NO SUITABLY DEREF. URI (8 queries):

> As discussed previously, dereferencing class and predicate URIs rarely returns their extension, and this is the case for DBpedia; for example, looking up the class `dbo:SoccerPlayer` does not return all instances of soccer-player, and looking up the predicate `dbo:thumbnail` does not return all relations between things and their thumbnails. A total of 8 query template instances would *only* involve URIs in such positions. (We do not include a further 4 `OPTIONAL` queries that also do not contain a suitably dereferenceable URI.)

This is perhaps an interesting observation in itself: only 9 of the 25 DBPSB query templates (36%) mined from real-world query logs could potentially be answered by LTBQE. If we were to relax the restriction on `OPTIONAL` and run it in a "best-effort" manner – or with a closed dataset semantics – LTBQE could run potentially run 13 of the 25 DBPSB queries (52%).

OVERVIEW OF EXPERIMENTS For the evaluation, we wanted to generate 25 sample queries for each of the remaining 9 DBPSB templates. For this, we ran the template queries provided for the benchmark against the public DBpedia SPARQL endpoint[15] and generated up to 1,000 results. From the template results, we randomly selected 25 to generate the query instances. Of the 9 templates, we encountered problems instantiating another 3 due to problems with DBPSB and the DBpedia endpoint itself:

COULD NOT GENERATE 25 INSTANCES (4 queries):

> We did not get 25 instances for 4 of the templates using the public SPARQL endpoint. Of these, 2 template queries repeatedly timed out and thus could not be instantiated. The other 2 query templates returned insufficient results to generate enough concrete queries: both queries generated only two instances due to the use of the predicate `dbpprop:redirect`, which returns only two triples from the public endpoint.

One of the queries that could not be instantiated involved `UNION` patterns such that it could be run and still generate results without the template query variable being instantiated, so we include this in the evaluation (DB17). As such, we are left with only 6 DBPSB templates that are usable for evaluating our methods.[16] We had problems with another template query: the query plan for DB24 was ordered in

15 `http://dbpedia.org/sparql/`

16 All template instances are available online: `http://code.google.com/p/lidaq/source/browse/queries/dbpsb.swj.25.tar.gz`

such a way that the only dereferenceable URI was in a class position. Hence we are only left with 5 runnable templates out of the available 25 in DBPSB, which again motivates the necessity for our own benchmark.

We benchmarked 6 of the 10 LIDAQ setups: based on experiences with unstable DBpedia queries in FedBench and some initial runs for DBPSB, we dropped configurations involving the dynamic collection of schema data as they increased the demand of sources from DBpedia and exacerbate unstable behaviour (*cf.* Table 4.6 and Table 4.7). We thus focus on evaluating CORE, CORE^{-}, SEEALSO, SAMEAS, RDFS$_s$ and COMB$_s$. Queries are run live and directly over dereferenceable DBpedia data *in situ*; some queries may also traverse links from DBpedia and find additional answers remotely.

Table 4.8 shows high-level statistics about the results retrieved for the query instances generated for each template. We show the number of queries which returned some results, divided by those considered to be benchmark stable (see Section 4.5.1.1) and those which were not; and we also show the number of queries for each template which did not return results. The templates that generated queries with empty results involved `UNION` patterns with shared template queries, which caused problems that we discuss later, particularly for DB4 (cf. Query 4.16).

Template	Total	Non-Empty		Empty
		stable	*unstable*	
DB1	25	23	2	0
DB4	25	0	12	13
DB5	25	0	13	12
DB13	25	24	0	1
DB17	25	24	1	0

Table 4.8: Statistics about stability per DBPSB query template

DETAILED RESULTS The results of the DBPSB experiments are given in Table 4.9 and Table 4.10 , with average measures given across all query instances. For each query template class, we now discuss the results. Herein, variables marked like "`%%var%%`" are template variables, which are instantiated to create concrete instances of queries.

```
SELECT DISTINCT ?var1
WHERE {
  %%var%% rdf:type ?var1 .
}
```

Query 4.15: [DB1]: Return the type(s) of a certain entity

Query 4.15 consists of only one triple pattern with a URI in the subject and predicate position (DBpedia does not contain blank nodes in the template variable position). LIDAQ results for instances of this template are listed in Table 4.9 and Table 4.10. We see that each query returns on average about 3 results per query without reasoning. We see that our reduced source selection optimisation works well (CORE^{-} and the extensions based on it), reducing the average number of HTTP GET requests from 8 to 2 (a single source lookup including redirect) and thus requiring only ~20% of the time taken for CORE (the additional lookups are for the predicate `rdf:type` and for the bound class URIs). Furthermore, we see that reasoning also increases the number of results at the cost of additional query time:

	Setup	Term		Results		Time (s)		First (s)	HTTP	Data	Inferred
		avg.	σ	*avg.*	σ	*avg.*	σ				
DB1	CORE	3.09	±4.66	3.09	±4.66	7.2	±9.4	2.4	8	3,034.74	–
	CORE^{-}	3.09	±4.66	3.09	±4.66	1.5	±0.7	1	2	6.3	–
	SEEALSO	3.09	±4.66	3.09	±4.66	1.6	±1	1.1	2	6.3	–
	SAMEAS	3.26	±5.06	3.26	±5.06	5.6	±8.4	1.5	3.13	203.87	143.83
	RDFS$_S$	6.57	±5.85	6.57	±5.85	10.8	±21.9	5.6	2	100.39	53.35
	COMB$_S$	6.87	±6.54	6.87	±6.54	11.8	±27.3	2.1	3.13	297.26	237.39
DB4	CORE	51	±73.56	24.75	±36.96	71.2	±103.5	71	147.5	5,427.75	–
	CORE^{-}	51	±73.56	24.75	±36.96	71	±103.1	70.7	139	2,739.5	–
	SEEALSO	51	±73.56	24.75	±36.96	71.6	±104.5	71.3	140.25	2,777.75	–
	SAMEAS	164.75	±209.84	174.5	±226.3	246.8	±327.7	246.4	801.5	104,864.25	94,473.25
	RDFS$_S$	51.75	±73.72	46.25	±72.74	108.9	±103.3	73.3	141	8,940	5,663.5
	COMB$_S$	165.5	±210.04	323.5	±444.02	291.8	±332.8	247.1	807.25	130,207.25	119,221.75

Table 4.9: Results for DBSPB queries DB1 and DB4

sameAs sees only a minor increase in results, but RDFS_S and Comb_S more than double the results by inferring additional types through sub-class, domain and range semantics. The extensions supporting `owl:sameAs` require looking up (on average) approximately one additional source.

```
SELECT ?var5 ?var6 ?var9 ?var8 ?var4
WHERE {
  { %%var%% ?var5 ?var6 .
    ?var6 foaf:name ?var8 .
  }
  UNION
  { ?var9 ?var5 %%var%% ;
      foaf:name ?var4 .
  }
}
```

Query 4.16: [DB4]: List the names of entities which are connected (in either direction) to the query entity.

Some of the generated instances for Query 4.16 failed to return results since they bind literals to the template variable due to the second mention of `%%var%%` in the object position. In such cases, these literals cannot be dereferenced and LTBQE cannot find results.[17] Table 4.9 and Table 4.10 show the average results for all queries. Though sameAs generates some additional results, it also instigates some unstable behaviour (*cf.* Table 4.8), where we see a large number of triples being retrieved and inferred, and where we see that the number of lookups triples. We can also see the benefit of Core^- in reducing the number of lookups vs. Core while not affecting results.

```
SELECT DISTINCT ?var3 ?var4 ?var5
WHERE {
  { ?var3 dbpp:series %%var1%% ;
        foaf:name ?var4 ;
        rdfs:comment ?var5 ;
        rdf:type %%var0%% .
  } UNION {
    ?var3 dbpp:series ?var8 .
    ?var8 dbpp:redirect %%var1%% .
    ?var3 foaf:name ?var4 ;
        rdfs:comment ?var5 ;
        rdf:type %%var0%% .
  }
}
```

Query 4.17: [DB5]: List the name and comments of a given series with a given type; or list the name and comments of a series with a given type that redirects to a given URL.

From Table 4.8, we encountered some similar behaviour for the instance queries of Query 4.17 as for the previous Query 4.16: some bindings for the template variable `%%var1%%` were again literals, leading to query instances for which no results could be found through LTBQE. From the detailed results in Table 4.9 and Table 4.10, although sameAs and Comb_S found additional results, they did so at the cost of causing unstable behaviour, increasing the number of HTTP lookups by a factor of $\sim 10\times$. Again we see the benefit of Core^- in reducing the number of lookups vs. Core while not affecting results.

17 SPARQL does allow literals in the subject position, though not allowed by RDF.

	Setup	Term		Results		Time (s)		First (s)	HTTP	Data	Inferred
		avg.	σ	*avg.*	σ	*avg.*	σ				
DB5	CORE	12.1	±13.74	7.7	±10.1	9.5	±4.3	5	21.7	689.4	–
	CORE^{-}	12.1	±13.74	7.7	±10.1	7.4	±4.2	3.8	13.7	187.4	–
	SEEALSO	12.1	±13.74	7.7	±10.1	7.5	±4.2	3.7	13.9	188.3	–
	SAMEAS	39.6	±41.8	993	±2,410.14	196.9	±381.3	24.6	133.4	104,446.5	113,258.6
	RDFS$_S$	12.1	±13.74	7.7	±10.1	45.4	±4.3	6.4	13.7	1,352.8	822.4
	COMB$_S$	39.6	±41.8	993	±2,410.14	200	±238.3	16	119.1	99,241.5	97,387.9
DB13	CORE	3	±0	2.67	±0.48	4	±2.5	1.9	8.08	261.42	–
	CORE^{-}	3	±0	2.67	±0.48	2	±2	1.6	2	13.46	–
	SEEALSO	3	±0	2.67	±0.48	2.5	±2.3	1.6	2	13.46	–
	SAMEAS	3	±0	15.96	±13.35	18.6	±3.6	2.1	7.5	1,487.12	1,140
	RDFS$_S$	3	±0	2.67	±0.48	6.7	±21.2	1.9	2	540.17	252.21
	COMB$_S$	3	±0	17.21	±13.49	23.4	±25.8	2.2	7.5	1,889.21	1,540.88
DB17	CORE	228	±0	114	±0	124.5	±0.5	121.7	237.92	4,679.75	–
	CORE^{-}	228	±0	114	±0	122.9	±3.5	5.4	234	4,420.33	–
	SEEALSO	228	±0	114	±0	143.3	±3	5.1	237.33	4,421	–
	RDFS$_S$	228	±0	114	±0	161.6	±4.3	5.1	234	125,408.25	68,798.75

Table 4.10: Results for DBSPB queries DB5, DB13 and DB17

```
SELECT *
WHERE {
  { %%var%% rdfs:comment ?var0 .
    FILTER (lang(?var0) = "en")
  }
  UNION
  { %%var%% foaf:depiction ?var1 }
  UNION
  { %%var%% foaf:homepage ?var2 }
}
```

Query 4.18: [DB13]: List English comments, depictions and homepages for an entity.

Much like Query 4.15, the star-shaped Query 4.18 is quite straightforward for the LTBQE approach as the results in Table 4.9 and Table 4.10 illustrate. Again the reduced source selection (CORE) shows benefits, returning all results, but reducing the amount of HTTP lookups: only one source needs to be retrieved (requiring two lookups including the redirect), and results take 2.67 seconds to process in this setup.

In this case, RDFS reasoning alone has no effect on the results, but increases the time by a factor of 3.4×. Support for `owl:sameAs` statements increases the result size by a factor of 6×, but at the cost of a 9.5× increase in time as an average of 5.5 additional sources are fetched. When `owl:sameAs` and RDFS support are combined in COMB_S, a few additional answers are found over SAMEAS alone.

```
SELECT DISTINCT ?var2 ?var3
WHERE{
  { ?var2 dcterms:subject %%var%%. }
  UNION
  { ?var2 dcterms:subject
            dbpcat:Prefectures_in_France . }
  UNION
  { ?var2 dcterms:subject
            dbpcat:German_state_capitals . }
  ?var2 rdfs:label ?var3.
  FILTER (lang(?var3)="fr")
}
```

Query 4.19: [DB17]: List the french labels for entities with the subject either German state capitals, prefectures in France or the query defined subject.

We updated Query 4.19 to use `dcterms:subject` instead of the predicate `skos:subject`.[18] However, the updated query template times out; hence we run the query without the first union clause containing the template variable. The results in Table 4.9 and Table 4.10 show that this query is more expensive to execute than star-shape queries with specific subject URIs. In particular, there are 99 prefectures listed for France and 15 German capital states, as well as the members of the category given by the template variable to dereference. Given the large number of documents accessed, we found that extensions following `owl:sameAs` links took too long to run for our experiments; hence these results are omitted.

CONCLUSION First, we notice that many of the DBPSB queries are unsuitable for LTBQE, and that we ended up only being able to run a small fraction of the original queries. Second, we generally found that CORE^- offers good performance

18 Although there are puzzlingly some `skos:subject` predicates in DBpedia, they are not used to relate entities to categories: `dcterms:subject` is now used in this case.

with respect to CORE, with minimal effect on results. Third, we found that RDFS_s only had a significant effect for the first query, asking for the types of a given entity. Fourth, we found that following `owl:sameAs` links on DBpedia invoked high overhead and unstable behaviour for 3 of the 5 queries, but also found various additional answers (though primarily aliases of result URIs). Ultimately, we conclude that LTBQE and its extensions (particularly those involving reasoning) struggle to cope with the complexity of DBPSB queries, which are designed to put materialised engines through their paces. As such, we study next the practicability of LTBQE for a broad range of simple queries and diverse data providers.

4.5.2.3 *QWalk Results*

Having looked at the FedBench evaluation containing eleven manually crafted queries answerable by LTBQE over a small number of real-world sources, and the DBPSB queries based on real-world query logs answerable (mostly) over DBpedia, we now look at the QWalk benchmark (Section 4.4.2), which automatically builds a large set of queries answerable over a wide range of real-world sources. For this, using random walk techniques over the BTC'11 corpus, we created 100 queries for each of the 11 elemental shapes of the QWalk benchmark, giving a total of 1,100 initial queries. As before, we then ran these live over remote sources in an uncontrolled setting using various configurations of LIDAQ.

QUERY TESTING We first wished to filter out queries that did not return any answers or that did not show benchmark stable behaviour.

To begin, for the entity query classes, we look at how many queries return empty results, how many return stable non-empty results suitable for comparison, and how many return unstable non-empty results (see Section 4.5.1.1). Our notion of stability is measured across all ten configurations of LIDAQ, including the dynamic schema import extensions. We also looked at the breakdown of stable/unstable/empty results turning off the dynamic schema import (*i.e.*, turning off RDFS_d, RDFS_e, COMB_d, COMB_e). The results are shown in Table 4.11. Though the stability of ENTITY-O and ENTITY-SO queries are not significantly affected, the number of stable queries for ENTITY-S queries more than halves. Furthermore, as we will see later, the dynamic import of schemata often requires over 10× the runtime of CORE, and over 5× the runtime of static schema equivalents. Due to problems with instability and long runtimes, and given the number of queries in the benchmark, we do not run the dynamic schema configurations for QWalk queries.

Template	**Total**	**Stable**	
		wo/dyn.	*w/dyn.*
ENTITY-S	100	60	27
ENTITY-O	100	57	53
ENTITY-SO	100	59	54

Table 4.11: Stable entity queries with and without dynamic schema extensions

Thus, considering only the six configurations CORE^-, CORE, SEEALSO, SAMEAS, RDFS_s and COMB_s, and for each query shape, Table 4.12 provides a breakdown of the total number of queries that return some results and exhibit stable or unstable behaviour, as well as the number of queries with no results. Typewritten numbers correspond to categories for HTTP server response codes encountered for queries with no results; the column "*mix*" indicates that there are at least two URIs with

different response codes and the column "*data*" indicates that the missing results are not related to URI errors and we assume that the remote data changed. We select only non-empty and stable queries for our comparison.

Class	Non-Empty		Empty					
	s.	*uns.*	*all*	*4XX*	*5XX*	*6XX*	*mix*	*data*
ENTITY-O	57	7	36	18	2	11	0	5
ENTITY-S	60	5	35	18	1	11	0	5
ENTITY-SO	59	9	32	17	2	8	1	4
O-PATH-2	62	4	34	16	5	8	1	4
O-PATH-3	35	25	40	19	3	18	0	0
S-PATH-2	66	2	32	17	2	11	0	2
S-PATH-3	51	7	42	18	1	20	0	3
STAR-0-3	67	6	27	14	0	10	0	3
STAR-1-2	62	2	36	21	2	12	0	1
STAR-2-1	70	5	25	11	3	9	1	1
STAR-3-0	66	15	19	12	0	4	0	3

Table 4.12: Summary of stable, unstable and empty queries for QWalk benchmark

DETAILED RESULTS We now look at the average measures for results across all (non-empty stable) queries per query class: we begin with entity queries, then progress to star queries and eventually to path queries. Detailed results for each of our measures can be found for reference in Table D.1–Table D.5 of Appendix D. Herein, we plot the total time and result sizes in bar plots, where we measure the ratio of the analogous figure for CORE$^-$ (which always returns the fewest results and should be the fastest). Again, absolute measures can be found in Table D.1–Table D.5. In general, we found a lot of variance and outliers in our results; hence we summarise results with bar plots which show the 50^{th}, 75^{th}, 90^{th} and 100^{th} percentiles of the result-sizes and times across the query classes, where the percentiles characterise how the majority of queries behaved, and what outliers occurred.

entity-*: GET GENERIC INFORMATION ABOUT A GIVEN RESOURCE

Entity queries have the most simple query shape and are used in a wide range of applications to gather all available information about a certain entity, *e.g.*, for the user interfaces of entity search engines. They are also often (but not always) used as a simple mechanism to support SPARQL `DESCRIBE` queries. Figure 4.7 presents the increase in time over CORE$^-$ for all other configurations across the three classes of entity queries, broken down by percentiles, with the x-axis presented in log-scale, where the 10^0 line indicates no change from CORE$^-$. Figure 4.8 analogously presents the increase in query results returned versus CORE$^-$.

We can see from the 50^{th} percentile in Figure 4.7 that the CORE configuration – which dereferences predicates, values for `rdf:type` and URIs bound to non-join variables – often requires significantly more time to process queries than CORE$^-$ across all three entity query classes, with the most severe case (on the 100^{th} percentile) taking almost eight times longer for ENTITY-S. Conversely, Figure 4.8 shows that CORE almost never returns additional results beyond CORE$^-$.

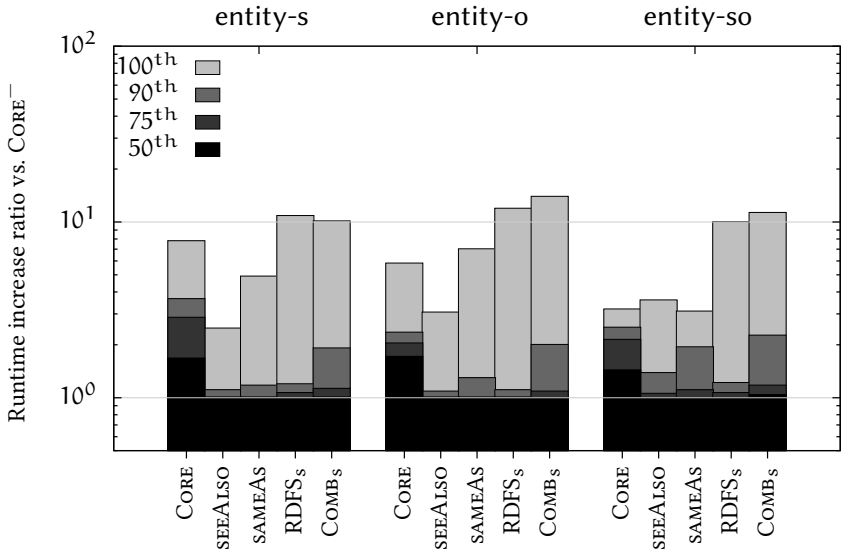


Figure 4.7: Percentiles for ratio of increase in runtimes vs. Core⁻ for entity-query classes (log)

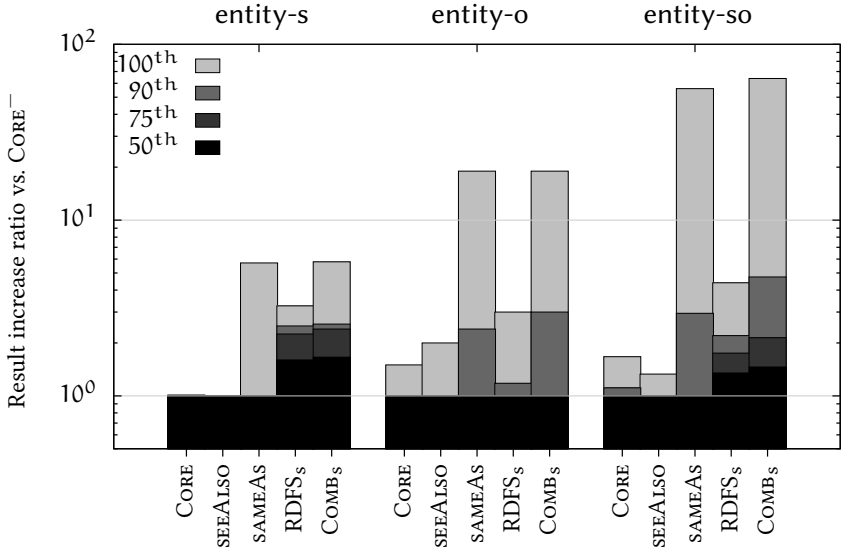


Figure 4.8: Percentiles for ratio of increase in results vs. Core⁻ for entity-query classes (log)

Similarly from both figures, we see that seeAlso rarely affects performance, but very rarely finds additional answers (only for the 100^{th} percentile are result increases visible). From the flat 75^{th} percentiles in Figure 4.7, we can see that in the majority of cases, other extensions did not affect performance significantly; however, the 90^{th} and 100^{th} percentiles show that reasoning can occasionally increase runtimes by a factor of over ten. However, reasoning can also increase result sizes by a large factor, where modest increases are visible already on the 50^{th} percentile

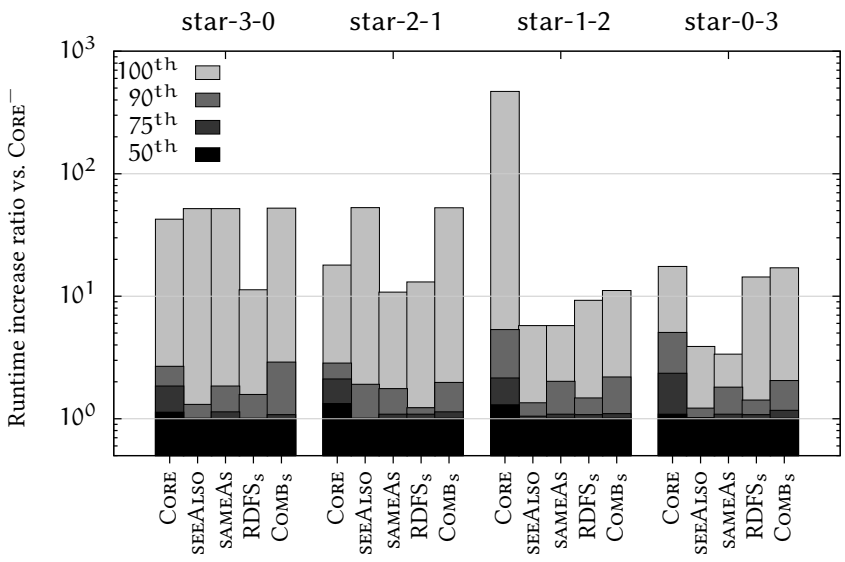


Figure 4.9: Percentiles for ratio of increase in runtimes vs. Core⁻ for star-query classes (log)

for RDFS_s and Comb_s in the entity-s and entity-so queries: all RDFS rules offer additional data for entity-s* queries, whereas only sub-property reasoning offers additional results for entity-o in the general case. Furthermore, the 75^{th}–100^{th} percentiles show occasional but very large increases for results in the sameAs configuration also.

star-*: Retrieve values for specific predicates about a given resource

Star-shaped queries are used to display select attributes of a resource useful in a certain context. The results for star-shaped queries follow the same format as before, where Figure 4.9 shows the average increase in query-time for each configuration over Core⁻, and Figure 4.10 shows the increase in result size.

Some similar conclusions can again be drawn as for entity queries. Again, the query time can often be reduced without a significant effect on query results by opting for Core⁻ over Core; in this case, since query predicates are set, the savings are primarily for not dereferencing URIs bound to non-join variables and values for `rdf:type`. The notable outlier for the query class star-1-2 in Core (on the 100^{th} percentile) is due to one query which took around 1 hour to terminate because of the download and processing of a very large document from the `ecowlim.tfri.gov.tw` provider (this source contributed no results and was not accessed by configurations built on top of Core).

Again, we see that seeAlso had minimal effect on results returned, but did increase the time significantly for some of the query classes. We also again see that RDFS and `owl:sameAs` reasoning has an occasional but significant effect on results size: however, we highlight that the baseline gave, on average, very few results for star-3-0 and star-0-3 (*cf.* Table D.2 and Table D.4), where a small absolute increase could account for a very large relative increase, as per the outliers on the 100^{th} percentile. Also, the large RDFS-related results outlier for the class star-1-2 is attributable to the query mentioned in 4.10.

-path-: Retrieve terms that are two or three hops away from a central resource through a path of given predicates

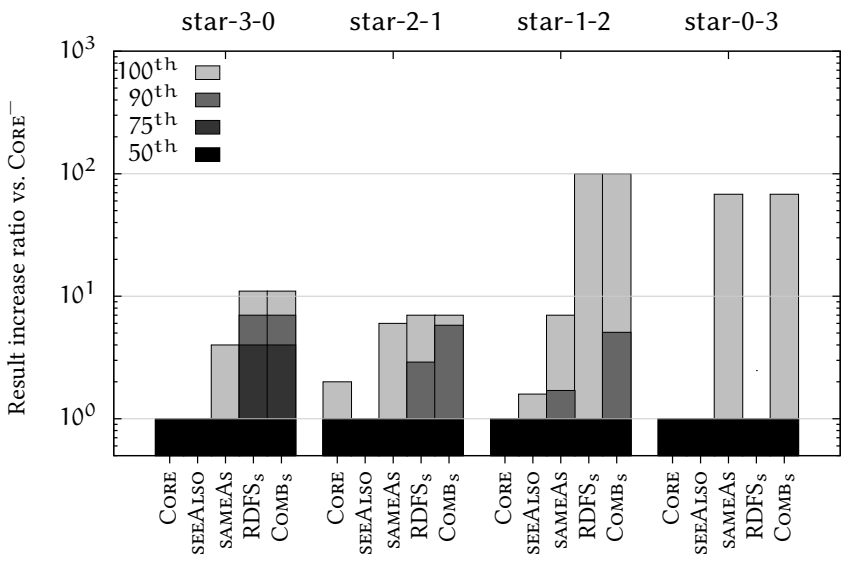


Figure 4.10: Percentiles for ratio of increase in results vs. Core⁻ for star-query classes (log)

Path-shaped queries allow for exploring recursive relations in the graph, or to discover particular information about neighbouring nodes. When compared with entity and star queries, we would expect path queries to generally be more expensive for LTBQE to process since they explicitly require traversing a number of sources.

For the six LiDaQ extensions, we again show the average increase of query time in Figure 4.11 and the average relative recall improvement in Figure 4.12. Again, we see the savings in time for selecting Core⁻ over Core, particularly for the o-path-* classes of queries. In general, across all extensions, the performance hits for the o-path-* queries are not met with gains in results; in fact, the o-path-3 queries saw no significant gains for any extension, even for the 100^{th} percentile. With respect to the QWalk results, we see the first meaningful gain for seeAlso in the s-path-3 class, but only for a single outlier query. In this case, sameAs offers only minimal increases in some outlier cases. However, the $RDFS_s$ extension does find additional results for s-path-* queries, which are notable already on the 75^{th} percentile; this extension performs particularly well for s-path-3 where large gains in results do not cost comparable increases in runtimes. The $Comb_s$ configuration again offers the most results, but – with the exception of Core– at the cost of the highest runtimes.

DISCUSSION Across the hundreds or queries run for the 11 query classes, we consistently find that the Core⁻ configuration saves significantly on query runtimes while not significantly reducing result sizes versus Core. With the exception of one query, we find that seeAlso finds barely any additional results, but can sometimes cause a significant increase in runtime. Reasoning extensions also increase runtimes, but regularly contribute additional answers: sameAs offers infrequent but very high increases in result sizes, where by comparison, $RDFS_s$ offers more frequent but more modest increases in results. These observations on result increases for the three extensions correspond well with the results of our analysis for the BTC'11 data in Section 4.3. Throughout, with the frequent exception of

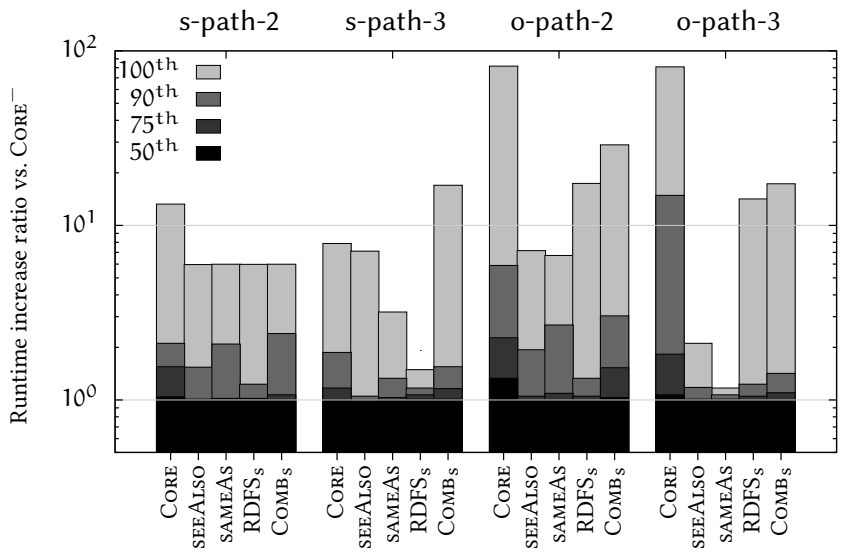


Figure 4.11: Percentiles for ratio of increase in runtimes vs. CORE⁻ for path-query classes (log)

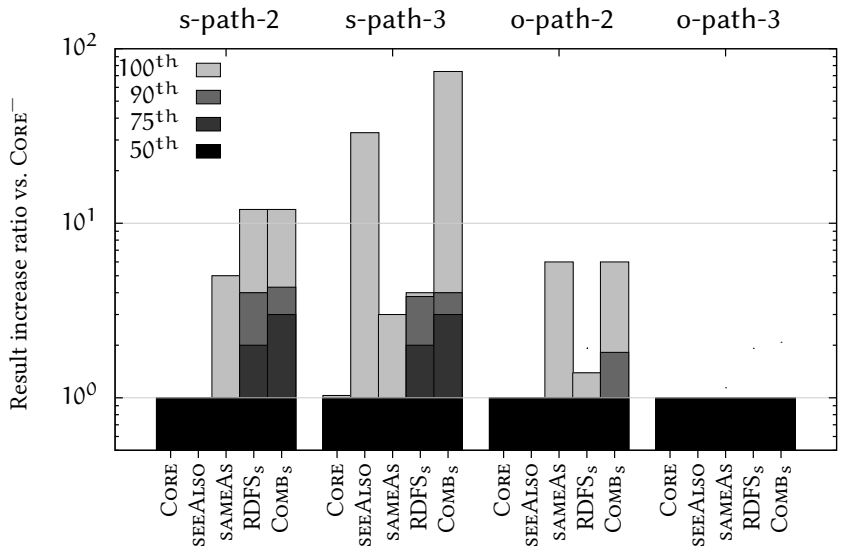


Figure 4.12: Percentiles for ratio of increase in results vs. CORE⁻ for path-query classes (log)

CORE, the combined approach was indeed the slowest, but always offered the most results.

To help summarise the potential benefit of each configuration, we present in Table 4.13 the average throughput (results per second) achieved across all queries per query class.[19] We see that CORE has uniformly the worst throughput of results

19 Given that there is a lot of variance in the raw figures, we acknowledge that average figures are a coarse way to present the results, but they do help to summarise overall trends.

	Core	Core$^-$	seeAlso	sameAs	RDFS$_s$	Comb$_s$
entity-s	1	1.68	1.67	2.15	1.29	1.53
entity-o	3.97	6.48	6.16	5.7	5.37	4.38
entity-so	2.02	2.82	2.66	3.71	3.73	4.82
star-3-0	0.11	0.16	0.15	0.15	0.24	0.2
star-2-1	0.58	1.12	1	1.04	2.14	1.75
star-1-2	0.17	1.6	1.35	1.6	70.97	58.85
star-0-3	0.18	0.35	0.33	0.94	0.24	0.68
s-path-2	0.44	0.72	0.68	0.7	0.83	0.78
s-path-3	1.76	2.45	2.56	2.46	2.43	2.1
o-path-2	1.38	8.39	7.76	10.55	6.36	6.89
o-path-3	0.95	5.7	5.84	6.08	5.04	4.68

Table 4.13: Results per second for all our query classes with configurations colour shaded from best (lightest) to worst (darkest) throughput

across all query classes. We also see that CORE$^-$ generally performs slightly above average, but performs best for ENTITY-O. With the sole exception of the S-PATH-3 class, SEEALSO performs slightly worse than CORE$^-$. In terms of the reasoning extensions, of the 11 query classes, the highest throughput for 9 are split between the SAMEAS (4), RDFS$_s$ (4) and COMB$_s$ (1) configurations, where, for each configuration, the throughput of COMB$_s$ frequently sits between SAMEAS and RDFS$_s$. However, aside from CORE, these latter configurations also often perform the worst: they add significant overhead to the query execution, but may often find significantly more additional results: they offer high-risk but high-gain.

4.6 CONCLUSION

In theory, proposed link traversal based query approaches for Linked Data have the benefit of up-to-date results and decentralised execution. However, in practice, a thorough evaluation of such methods in realistic uncontrolled environments – for a diverse Web of Data – had not yet been conducted. This chapter focused on evaluating LTBQE approaches in this manner, and similarly investigated the possibility of combining lightweight reasoning methods with LTBQE to help squeeze additional answers from query relevant sources, and to help integrate data from diverse data-providers.

We have characterised what percentage of data is missed by only considering dereferenceable information, we have looked at what percentage of raw data is made available to LTBQE through various extensions. We have tested LTBQE and various extensions in uncontrolled environments for three complimentary query benchmarks. Our results show that LTBQE works well for simple queries with a dereferenceable subject, but, in uncontrolled environments, struggles for more complex queries that involve accessing many remote sources at runtime. Furthermore, we showed that runtimes in uncontrolled environments are often a factor of politeness policies, since queries often touch upon documents from the same domain.

In terms of the extensions, we have shown that the selection of sources can be successfully reduced by ignoring predicate URIs, object URIs for type-triples, and URIs bound to non-join positions. We have also shown that the `rdfs:seeAlso` extension offers little in terms of results, but occasionally introduces significant runtime costs. Furthermore, the `owl:sameAs` extensions can occasionally increase the number of results found by a great deal, but also comes at significant costs

and introduces unstable behaviour when run live over domains such as DBpedia. Similarly, our comprehensive evaluation showed that RDFS reasoning extensions increase results more frequently than `owl:sameAs` extensions (*e.g.*, in lower percentiles of the QWalk experiments), but exhibits more moderate increases than the latter extensions (*e.g.*, in the 100^{th} percentiles of QWalk experiments). The FedBench experiments showed that the dynamic import of RDFS data at runtime works well for simple queries on certain domains (*e.g.*, `data.semanticweb.org`), but can introduce instability for domains such as DBpedia, where schemata are spread across multiple documents and link to other domains with similar decentralised schema.

Taking an alternative view, LTBQE is an interesting *technique* for SPARQL and is complementary to other techniques for querying Linked Data, such as materialised or federated approaches. LTBQE offers the potential to get fresh answers when dynamic information is involved, or to get sensitive data when user-specific access-control is in place for some Linked Data source; this is not possible through centralised approaches. Furthermore, it does not rely on SPARQL interfaces like federated approaches; also, there are currently no mechanisms to *discover* endpoints in the same manner that LTBQE discovers sources.

As such, we see the greatest potential for LTBQE in combination with other querying techniques for example to dynamically freshen-up results returned by a centralised SPARQL endpoint that replicates remote content, as we will describe in Chapter 6

In general, in absence of local knowledge about Web source, the types of queries that LTBQE can answer are severely limited (reliant on dereferenceable URIs). Hence, in the following chapter, we investigate a novel query ing method that uses local data summaries to overcome this limitation of LTBQE.

5 COMPARISON OF SOURCE SELECTION METHODS

In the previous chapter, we showed that link-traversal based query execution approaches perform well for simple queries, especially if query patterns contain dereferenceable subject URIs. However, the fundamental drawback of LTBQE is that query time and result recall depends on the sources which are discovered in a bottom-up fashion during the query execution. This source discovery process is limited and specified primarily by the query itself, the execution order and the links between query relevant data.

In this chapter, we propose a different but complementary approach. We study and compare index structures which are used to identify query relevant sources in a top-down fashion before the actual query execution.

We propose and investigate the use of "data summaries" for determining which sources contribute answers to a query based on an approximate multidimensional hash based indexing structure (a QTree [Hose et al., 2005]). These data summaries concisely describe the contents of a large number of sources from the Web of Data and form the basis for (i) *source selection* prior to query execution by ruling out sources that cannot contribute to the overall query or joins within (sub-)queries, and (ii) *query optimisation* to provide the optimal join order and determine the most important sources via ranking.

The remainder of this chapter is structured as follows:

- we introduce our generic query processing model and present several source selection approaches in Section 5.1;
- we detail two possible approaches for data summarisation and source selection based on multidimensional histograms and QTrees, including the constructing and maintaining these data summaries in Section 5.2;
- we discuss different alternatives for hashing, i.e., the translation from identifiers to a numerical representation that both data summaries rely upon in Section 5.3;
- we present our algorithms for source selection, join processing, and discuss source ranking in Section 5.4.
- we present the experimental setup for our evaluation in and discuss in detail the experimental results of comparing different data summaries and hashing alternatives in Section 5.5;
- we conclude and provide an outlook to future work in Section 5.6.

5.1 SOURCE SELECTION APPROACHES

In this section we present our generic query processing model and introduce several source selection approaches, including schema-level and inverted indexes and our data summaries.

5.1.1 Generic Query Processing Model & Assumption

We focus in this chapter on processing conjunctive SPARQL queries directly over Linked Data by discovering query relevant sources with index structures containing knowledge about the source content. In this process, the main problems are (i) to find the right sources which contain possible answers that can contribute to the overall query and (ii) the efficient parallel fetching of content from these sources.

As such, query processing in our approach works as follows:

1. prime a compact index structure with a seed data set (various mechanisms for creating and maintaining the index are covered in Section 5.2);
2. use the data-summary index to determine which sources contribute partial results for a conjunctive SPARQL query Q and optionally rank these sources;
3. fetch the content of the sources into memory (optionally using only the top-k sources according to auxiliary data for ranking stored in the data summary);
4. perform join processing locally, i.e., we do not assume that remote sources provide functionality for computing joins.

ASSUMPTIONS We see the presented query processing strategy as a reasonable compromise to a centralised index containing all source information under the following assumptions:

- The overall data distribution characteristics within different sources do not change dramatically over time, and can be captured in a data summary in lieu of a full local data-index. Our Linked Data dynamicity experiment in Section 3.2 showed that most of the monitored changes are updates.
- Source selection and ranking can reduce the amount of fetched sources to a reasonable size so that content can be fetched and processed locally.

We believe these assumptions hold for a wide range of Linked Data sources and typical queries and which we partially verify in Section 5.5.2

We illustrate this query processing approach based on the following examples, for which we consider that a source selection index is primed with the sources from our example graph in Figure 2.1 (as per step one in the above highlighted approach). This *source-selection* method takes as input a triple pattern and returns a list of data sources that potentially contribute bindings.

Example 5.1. Our first example illustrates the general process of our query approach with our introduction Query 2.2 asking for the names of the author for a certain publication. The query engine uses the source selection index to determine sources which potentially can contribute results for the single query pattern. Given our example graph, a straight forward source selection algorithm would determine the query relevant sources by retrieving and combining the list of sources for each query pattern. As such, the list of query relevant sources contains for the triple pattern ("`dblpP:HartigBF09 foaf:maker ?author .`") the three sources `dblpPDoc:HartBF09`, `dblpADoc:Olaf_Hartig` and `dblpPDoc:Christian_Bizer`. The three sources `ohDoc:`, `chDoc:` and `dblpPDoc:Christian_Bizer` contribute answers to the second triple pattern ("`?author foaf:name ?authorName .`"). In the next step, the engine collects in parallel the content of the selected sources and evaluates the query over the merged data graph.

The next example illustrates how the top-down source selection query approach can potentially answer more queries compared to the LTBQE approach.

Example 5.2. We discussed in Chapter 4 that the LTBQE approach cannot successfully execute our example Query 4.4 (which asks for people with the same name) since the two query patterns are joined by non dereferenceable RDF literal values. However, consider that a query engine has an source selection index that contains, from the first step in our query processing approach, the information about the five documents in our example graph (cf. Figure 2.1). In the seconds step, the query engines uses the index to determine the list of sources which contribute to the two query patterns of the query, resulting in `ohDoc:` and `dblpADoc:Olaf_Hartig`. Given that list, the query processor retrieves the content of the sources and evaluates the query to get the list of people that have the same names.

As we can foreseen from our simple examples, a potentially large number of sources can contribute bindings to each of the triple patterns. Since accessing too many sources over the Web is potentially very costly, the choice of the source-selection method is fundamental, and strongly depends on the various possible query approaches and their underlying index structures, which we discuss next.

5.1.2 Source-Selection Approaches

In the following we introduce approaches for source selection, starting with the approach that introduces the least complexity, and then describe each approach in more detail. We do not cover standard RDF indexing approaches (i.e., complete indexes [Harth and Decker, 2005; Neumann and Weikum, 2008, 2010; Weiss et al., 2008]) as our premise is to perform query processing over data sources directly: rather than maintaining complete indexes and deriving bindings from index lookups, we aim at using data structures that just support the selection of data sources from which data will be retrieved and then processed locally. That is, the general idea is to use index structures to guide the query processor, while being reasonable both wrt. completeness of the answers and network bandwidth consumption. Possible approaches to evaluate queries over Web sources and particularly addressing the problem of source selection are:

LTBQE:

> as discussed in detail in the previous chapter, this approach exploits the correspondence between resource URIs mentioned in the query and source URI. That is, only URIs mentioned in the query or URIs from partial results fetched during query execution are looked up directly without the need for maintaining local indexes. Since the source URIs can be derived from the URIs mentioned in the query, the approach does not need any index structures, but – given the structure of Linked Data and our results in Chapter 4– will likely only return answers for selected triple patterns (see Table 5.1).

SCHEMA-LEVEL INDEXING (SLI):

> relies on schema-based indexes known from query processing in distributed databases [Goldman and Widom, 1997; Stuckenschmidt et al., 2004]. The query processor keeps an index structure of the schema, i.e, which properties (URIs used as predicates in triples) and/or classes (i.e., objects of `rdf:type` triples) occur at certain sources and uses that index to guide query processing. Triple patterns with variables in predicate position cannot be answered (see Table 5.1).

INVERTED URI INDEXING (II)

indexes all URIs occurring in a given data source similar to inverted document indexes in search engines. An II covers occurrences of URIs in sources. The II allows the query processor to identify all sources which contain a given URI and thus potentially contribute bindings for a triple pattern containing that URI. Using an inverted URI index, a query processor can obtain bindings from sources which contain the pertinent URI but for which the resource/data source correspondence as specified in the Linked Data principles does not hold.

Example 5.3. Taking our example Figure 2.1, the source `ohDoc:` contains additional descriptions about `cb:Chris` not mentioned in `cbDoc:` like an addition image or contact information not provided in `cbDoc:`. In our example, the additional statement is the `rdfs:seeAlso` information which links the non-dereferenceable URI to its document. The index stores such mentions of URIs outside their implicitly associated source.

MULTIDIMENSIONAL HISTOGRAMS (MDH):

combine instance- and schema- level elements to summarise the content of data sources using histograms [Hose et al., 2009; Petrakis et al., 2004]. MDHs represent an approximation of the whole data set to reduce the amount of data stored. We will present one type of MDH in more detail in Section 5.2.

QTREE (QT):

The QTree [Hose et al., 2005] is another approach that uses a combined description of instance- and schema-level elements to summarise the content of data sources. In contrast to the MDH where regions are of fixed size, the QTree is a tree-based data structure where regions of variable size more accurately cover the content of sources.

If we consider all possible combinations of constant and variables in triple patterns in the BGPs of SPARQL queries, we realise that different source selection mechanisms only cover a subset of those. At an abstract level, triple patterns can have one of the following eight forms (where ? denotes variables and # denotes constants):

```
(?s ?p ?o) (#s ?p ?o) (?s #p ?o) (?s ?p #o)
(#s #p ?o) (#s ?p #o) (?s #p #o) (#s #p #o)
```

Table 5.1 lists the triple patterns that can be answered using the respective source-selection mechanism. For example, we showed in Chapter 4 that the LTBQE approach cannot find answers to triple pattern with only variables, but performs reasonably good for patterns with subject URIs. Schema level indexes (SLI) can only be used for patterns which contain predicates or `rdf:type` objects.

Which source-selection approach to use depends on the application scenario. LTBQE works without additional index structures, but fails to incorporate URI usage outside the directly associated source (via syntactic means – the URIs containing a # – or via HTTP redirects). Further, LTBQE follows an inherently sequential approach of fetching relevant sources (as new sources are discovered during query execution), whereas the other mentioned indexing approaches enable a parallel fetch of relevant sources throughout. II can leverage optimised inverted index structures known from Information Retrieval, or can use already established aggregators (such as search engines) to supply sources. While not supporting full joins,

Approach	Triple Patterns
LTBQE	`(#s ?p ?o)`, `(#s #p ?o)`, `(#s #p #o)`
	and possibly `(?s ?p #o)`, `(?s #p #o)`
SLI	`(?s #p ?o)`, `(?s rdf:type #o)`
II, MDH, QT	all

Table 5.1: SPARQL triple patterns supported by different source selection approaches

II has been extended to support simple, star-shaped joins [Delbru et al., 2012, 2010]. SLI works reliably on queries with specified values at the predicate position, and can conveniently be extended to indexing arbitrary paths of properties [Stuckenschmidt et al., 2004]. However, both II and SLI, are "exact" indexes which have the same worst-case complexity as a full index, i.e., potentially grow proportionally to the number and size of data sources (or, more specifically with the number URIs mentioned therein).[1]

In the present chapter, we will particularly focus on the deployment of approximate data summaries which can be further compressed depending on available space – at the cost of a potentially more coarse-grained source selection results – namely MDH and QT, which only necessarily grow with the number of data sources, but not necessarily with the number of different URIs mentioned therein. MDH is an approach inexpensive to build and maintain but may provide a too coarse-grained index which negatively affects the benefit of using the index. QT is more accurate, as the data structure is – as we will see – able to represent dependencies between terms in single RDF triples and combinations of triples, which can be leveraged during join processing, however, at increased cost for index construction and maintenance. Note that approaches such as II and SLI do not model those dependencies, which can result in suboptimal performance.[2]

We provide detailed experimental evaluation for each of the mentioned index structures in Section 5.5.2.

5.2 DATA SUMMARIES

In general, data summaries allow the query processor to decide on the relevance of a source with respect to a given query. Querying only relevant instead of all available sources can reduce the cost of query execution dramatically.

We use the data summaries to describe the content of Web sources in much more detail than schema-level indexes which, in general, reduces the number of queried sources but also allows for more triple patterns. Data summaries represent the sources data in an aggregated and compact form. As summarising numerical data is more efficient than summarising strings, the first step in building a summary index is to transform the RDF triples provided by the sources into numerical space. We apply hash functions to the individual terms of RDF triples (`s, p, o`) to maps a triple of string values to a triple of numerical values (numbers). The resulting "numerical" triples are inserted in to the data summary together with the source identifier the statement originate from. It is necessary to attach the source

1 While this might be viewed as a theoretical limitation not relevant in practical data, we will also see other advantages of alternative data summaries that we focus on in this paper.

2 While the mentioned extensions of SLI [Stuckenschmidt et al., 2004] or extensions of II [Delbru et al., 2012, 2010] partially address this issue as well, they do not cover arbitrary acyclic queries, but only fixed paths in the case of [Stuckenschmidt et al., 2004] or star-shaped queries in the case of [Delbru et al., 2012, 2010].

information for each numerical triple since we use the summary to obtain a set of sources potentially providing relevant data for a given query. To do so, we will discuss in Section 5.3 how we map the query patterns to their numerical representation and in Section 5.4 how we probe the index for the relevant sources.

In the remainder of this section, we introduce the two variants of data summaries we focus on, namely, multidimensional histograms [Ioannidis, 2003] and QTrees [Hose et al., 2005, 2006; Zinn, 2004]. We discuss for index insertion and lookup the time and space complexities of the operations and give details on how to initialise and expand data summaries in general.

5.2.1 Multidimensional Histograms

Histograms are one of the basic building blocks for estimating selectivity and cardinality in database systems. Throughout the years, many different variations have been developed and designed for special applications [Ioannidis, 2003]. However, the basic principles are common to all of them. First, the numerical space is partitioned into regions, each defining a so-called *bucket*. A bucket contains statistical information about the data items contained in the respective region. If we need to represent not only single values but instead pairs, triples or n-tuples in general, we need to use multidimensional histograms with n dimensions. In n-dimensional data space, the regions represented by the buckets correspond to n-dimensional hypercubes. For simplicity, however, we refer to them simply as regions. For RDF data, we need three dimensions – for subject, predicate, and object.

Data items, or triples respectively, are inserted one after another and aggregated into regions. Aggregation and thus space reduction is achieved by keeping, for each three-dimensional region, statistics instead of the full details of the data items. A straightforward option is to maintain a number of data items per region and a list of sources contributing to this count. However, we show that source selection and ranking perform better if we maintain a set of pairs (`count`,`source`) – denoting that `count` data items provided by the source `source` are represented by the region.

The main difference between the histogram variations is how the bucket boundaries are determined. There is a trade-off between construction/maintenance costs and approximation quality. Approximation quality is determined by the size of the region and the distribution of represented data items – on big region for only a few data items has a higher approximation error than several small regions for the same data. The quality of a histogram also depends on the inserted data. We decided to use equi-width histograms as an example representative for MDH, because they are easy to explain, apply to a wide range of scenarios and can be built efficiently even if the exact data distribution is not known in advance.

For this kind of histograms, given the minimum and maximum values of the numerical dimensions to index and the maximum number of buckets per dimension, each dimensional range is divided into equi-distant partitions. Each partition defines the boundaries of a region/bucket in the dimension. The upper part of Figure 5.1 shows a two-dimensional example of a multidimensional equi-width histogram with the number of buckets per dimension set to three.

Given an RDF triple, a lookup entails computing the corresponding numerical triple by applying the same hash function to the RDF triple that has been used for constructing the histogram and retrieving the bucket responsible for the obtained numerical triple. The bucket contains information about which sources provide how many data items in the bucket's region. Hence, we only need to consider the found relevant sources. However, there is no guarantee that the sources actually provide the RDF triple that we were originally looking for (false positives). The reason is that a bucket does not represent exact coordinates in data space but a

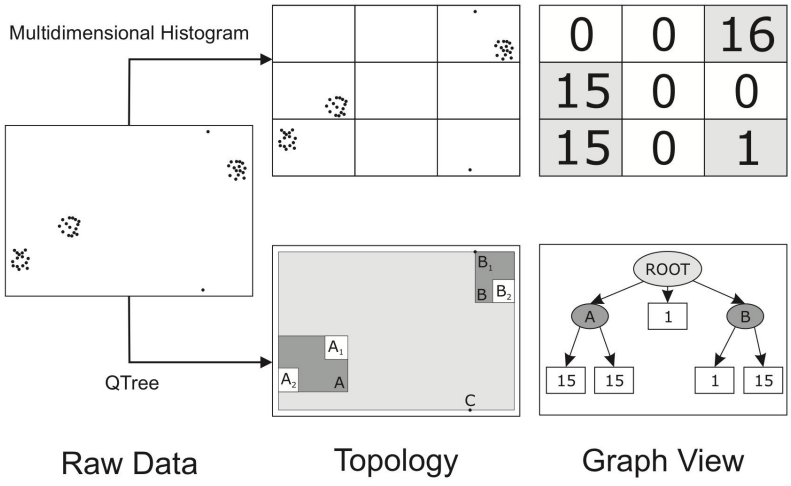


Figure 5.1: Example of a data summary. The left column shows the coordinates corresponding to hash values for the data items to insert. The middle column shows the bucket regions, and the right column shows the buckets with the assigned statistical data and, in case of the QTree, the hierarchy between inner nodes and buckets.

data summary	*space complexity*
MDH	$O(b_{max} \cdot c_{src})$
QTree	$O(b_{max} \cdot c_{src})$

Table 5.2: Space Complexity of QTrees and Multidimensional Histograms.

region which also covers coordinates for RDF triples not provided by any indexed source.

Space Complexity

Determining space consumption for the histogram is straightforward; denoting the total number of buckets used by a multidimensional histogram as b_{max}, the number of sources by c_{src}, and considering the maximum number of (`count`,`source`) pairs per bucket, we can state that it requires $O(b_{max} \cdot c_{src})$ space (Table 5.2).

Runtime Complexity

For the multidimensional histogram introduced above, determining the bucket that a numerical triple d has to be inserted into has complexity $O(1)$ – having arranged the buckets in an array, the coordinates of the searched bucket can easily be determined based on the static boundaries of the regions. The insertion itself can be done in $O(\log c_{src})$ by finding the bucket's (`count`,`source`) pair corresponding to d's source and adapting the `count` value – using a Java TreeMap to manage the pairs. A lookup resembles the procedure of determining the bucket responsible for the data item to insert and is therefore also done in $O(\log c_{src})$ (Table 5.3).

5.2.2 QTree

The QTree – originally developed for top-k query processing in P2P systems [Hose et al., 2005, 2006; Zinn, 2004] – is a combination of multidimensional histograms

	Data summary	*Time Complexity*
INSERT	MDH	$O(\log c_{src})$
	QTree	$O(b_{max} \cdot f_{max}^2 + c_{src} \cdot \log c_{src})$ or $O(\log b_{max} \cdot f_{max}^2 + c_{src} \cdot \log c_{src})$
LOOKUP	MDH	$O(\log c_{src})$
	QTree	$O(b_{max} + \log c_{src})$ or $O(\log b_{max} + \log c_{src})$

Table 5.3: Time Complexity of QTrees and Multidimensional Histograms.

and R-trees [Guttman, 1984] and therefore inherits benefits from both data structures: indexing multidimensional data, capturing attribute correlations, efficiently dealing with sparse data, allowing efficient look-ups, and supporting incremental construction and maintenance.

In contrast to the histograms introduced above and similar to R-trees, QTrees are hierarchical structures. They consist of nodes representing regions in the data space. The region of a node always covers all the regions of its child nodes. Data items are only represented by leaf nodes – in analogy to the multidimensional histogram introduced above we refer to leaf nodes as buckets and store the same information in them. The lower part of Figure 5.1 shows an example QTree with all regions of inner nodes and buckets as well as the hierarchy between them.

In contrast to standard histograms, QTrees do not necessarily cover all the data space but only regions containing data. Thus, in case of sparse data the histograms introduced above use same sized regions for areas representing many data items as well as for areas containing only a few data items. QTrees, however, use regions of variable sizes covering only areas containing data. The number of nodes in a QTree is determined by two parameters: b_{max} denoting the maximum number of buckets in the QTree and f_{max} describing the maximum fanout (i.e., the number of child nodes) for each non-leaf node.

Although R-trees and QTrees share the same principle of indexing data by organising multidimensional regions hierarchically, they differ in a substantial way: whereas the QTree approximates the indexed data to reduce space consumption, R-trees keep detailed information about inserted data items (tuples in our case). A QTree's leaf node provides statistical information about the tuples it represents, i.e., the number and origin (source) of tuples located in the multidimensional region – and no information about the tuples' coordinates. In contrast, an R-tree's leaf node keeps the exact coordinates of the tuples and therefore consumes more space than a QTree. Thus, space consumption for QTrees (maximum number of buckets determines the degree of approximation) increases only with the number of data sources whereas space consumption for an R-trees variant holding the same information increases both with the number of sources and the number of tuples.

When additional tuples are inserted into an R-tree, having access to the exact coordinates of all represented tuples allows for efficient re-balancing so that we can guarantee that the R-tree is always balanced, i.e., all leaf nodes are on the same level. As a consequence of approximation, we cannot guarantee balance for the QTree; as we do not know the exact coordinates of tuples represented by a particular region, we cannot simply split the region for re-balancing without losing accuracy/correctness. For splitting we would have to "guess" (due to the approximation) which of the resulting regions the original data should be assigned to. Therefore, instead of re-balancing algorithms like the R-tree, the QTree applies heuristics to keep itself

balanced. Although balancing works well in practice, we cannot give any guarantees because it is theoretically possible to construct a QTree that conforms to a linear list [Hose, 2009; Zinn, 2004].

QTrees are constructed by inserting one data item after another. However, due to the hierarchical structure, construction is more complex and adheres to the following steps (more details [Hose, 2009; Zinn, 2004]):

(1) TRY TO INSERT d INTO AN EXISTING BUCKET.

For each data item d, we first check whether d can be added to an existing bucket whose region encloses d's coordinates. If found, the bucket statistics are updated by adapting the (`count`,`source`) pair corresponding to the source that d originates from. If a pair corresponding to the source already exists, we increase its `count` value by one. If such a pair does not exist, we add a new pair (1,`source`).

(2) FIND MOST RESPONSIBLE INNER NODE AND INSERT d AS A NEW BUCKET.

If d could not be inserted into an existing bucket, we traverse the QTree beginning at the root node. We look for a child node p whose region completely encloses d and proceed recursively with p. We stop when we cannot find such a child node. Instead, we create a bucket as a new child node with a single (1,`source`) pair representing d.

(3) ENFORCE f_{max} CONSTRAINT.

The previous step might have violated the f_{max} constraint by creating a new bucket and attaching it as a child to an inner node p. In order to reduce p's fanout, we determine the pair of p's child nodes whose merging would result in a node with the smallest region. Either we create a new child node n of p and attach the pair of chosen siblings as child nodes to n, or if the bucket inserted in the previous step is part of the pair of chosen siblings, we assign the node as a new child of its sibling node. The latter case might result in the need to recursively enforce the f_{max} constraint. In addition, we apply some heuristics to prevent the QTree from degenerating into a linear list by trying to destroy inner nodes whose children could be attached to the parent node without violating the constraints.

(4) ENFORCE b_{max} CONSTRAINT.

If step (3) resulted in a situation where the maximum number of buckets b_{max} is exceeded, we need to reduce the number of buckets in the QTree. Hence, we search for the pair of sibling buckets whose merging would result in a bucket with the smallest region. The so-found pair of buckets is merged into a new bucket whose region minimally encloses those of the original buckets and whose set of (`count`,`value`) pairs originates from merging pairs of the original buckets – pairs referring to the same source are merged by summing up their `count` values. Since the parent of the merged siblings now has less child nodes, we try to remove the parent by attempting to assign its child nodes to its parent.

In order to avoid comparing all sibling buckets of all levels to each other each time we need to enforce the constraints, we maintain a priority queue containing at most one entry for each inner node stating its pair of child buckets to be merged next. The entry corresponding to an inner node only needs to be updated when its child buckets are added or merged.

A lookup for an RDF triple in the QTree is very similar to the lookup in multi-dimensional histograms. The only difference is finding a bucket that contains the

numerical triple corresponding to the given RDF triple. In contrast to the histogram approach introduced above, buckets in a QTree might overlap so that we find multiple buckets for one given RDF triple. As we do not know into which one the triple has been inserted when constructing the index, we need to find all such buckets. We find these buckets by traversing the QTree starting at the root node and recursively following all paths rooted by children whose regions contain the coordinates defined by the numerical triple we are looking for – as regions are allowed to overlap, we might need to traverse multiple paths. Traversing a path ends at a bucket or when there are no further child nodes containing the coordinates. Just as for histograms false positives are possible, i.e., the index indicates the relevance of a source although in fact it does not provide the data item we were looking for.

Space Complexity

The QTree's space complexity comprises the space consumption of its main components: leaf nodes, inner nodes, and the priority queue. Note that the size of a QTree depends solely on its parameters (f_{max} and b_{max}) as well as the number of sources c_{src}, but is independent of the number of represented data items.

By enforcing the b_{max} constraint, we can ensure that a QTree contains at most b_{max} leaf nodes. Only leaf nodes hold statistical information about the sources in the form of (`count`,`source`) pairs. The size of each pair is fixed and the number of pairs depends on the number of sources c_{src}. Thus, leaf nodes require $O(b_{max} \cdot c_{src})$ space.

Construction ensures that an inner node has at most f_{max} and at least two child nodes. A QTree is a tree structure where each inner node has exactly two children and thus a QTree has at most $b_{max} - 1$ inner nodes. Allowing inner nodes to have more children, i.e., f_{max} which is always greater than or equal to 2, never increases but only reduces the number of inner nodes. Thus, a QTree has at most $b_{max} - 1$ inner nodes. As the priority queue has at most one entry for each inner node, it cannot have more than b_{max} entries. Consequently, a QTree requires $O(b_{max})$ space for inner nodes and the priority queue. Hence, in total a QTree requires $O(b_{max} \cdot c_{src})$ for its main components altogether (Table 5.2).

The space complexity of both, multidimensional histograms and QTrees, highlights an earlier-mentioned advantage of these hash-based data summaries in comparison to other indexing approaches discussed in Section 5.1.2: the other indexes sketched there grow in the worst case with the number of indexed triples. In contrast, due to the supported adaptive approximation, histograms and QTree grow only with the number of data sources, independent from the number of indexed triples. Moreover, the total size of the data summary is adjustable by setting an appropriate b_{max} value.

Runtime Complexity

To determine runtime complexity for constructing the QTree, we need to determine the costs for each main step of the insertion algorithm – we omit proofs and refer interested readers to [Zinn, 2004] for more details. When inserting a data item d, these costs are:

(1) Try to insert d into an existing bucket
    $O(b_{max} + \log \texttt{sources})$
(2) Find most responsible inner node and insert d as a new bucket
    $O(b_{max})$
(3) Enforce f_{max} constraint

$O(b_{max} \cdot f_{max}^2)$

(4) Enforce b_{max} constraint

$O(f_{max}^2 + \log\ b_{max} + c_{src} \cdot \log c_{src})$

In the first step, we try to insert d into an existing bucket. In the worst case, we might have to traverse all buckets, thus $O(b_{max})$ time. Updating all the (`count`, `source`) pairs requires $O(\log$ `sources`$)$ time using a Java TreeMap. Thus, time complexity is $O(b_{max} + \log$ `sources`$)$ in total.

In the second step, we traverse the tree until we find a node at which we can insert d as a new bucket. In the worst case, we have to visit all nodes and check if d is contained in the nodes' regions: $O(b_{max})$. The insertion of d as a new bucket takes constant time $O(1)$.

The third step is more expensive as we need to enforce the f_{max} constraint possibly recursively through the tree. In the worst case, we need to make adaptations to each level in the tree, i.e., enforcement is required at most b_{max} times. For each call we need to find that pair of child nodes whose merging would result in the smallest region. Thus, we need to compare each pair of child nodes, i.e., $O(f_{max}^2)$. Rearranging child nodes and creating a new inner node takes $O(f_{max}^2)$ time. Thus, in total the third step requires $O(b_{max} \cdot f_{max}^2)$ time.

In the fourth step, we need to enforce the b_{max} constraint and reduce the number of buckets. Finding the best pair of buckets is done in $O(1)$ because all we have to do is to remove the first entry from the priority queue. Merging two buckets requires $O(c_{src} \cdot \log c_{src})$ using the Java TreeMap to organise (`count`,`source`) pairs. The priority queue needs to be updated with respect to the parent node of the merged buckets – finding the best pair of buckets takes $O(f_{max}^2)$ and updating the priority queue takes $O(\log\ b_{max})$ operations using a heap-based priority queue. In any case, whether the parent node can be dropped – $O(f_{max}^2)$ – or not, only one entry of the priority queue needs to be updated. Thus, in total step four takes $O(f_{max}^2 + \log\ b_{max} + c_{src} \cdot \log c_{src})$ time.

Performing a lookup operation in the QTree requires at most $O(b_{max} + \log c_{src})$ because in the worst case we need to visit all nodes, and lookups in the (`count`,`value`) pairs cost at most $O(\log c_{src})$.

The above runtime complexities are based on the worst case corresponding to an unbalanced QTree with a tree height of b_{max}. The construction algorithm cannot guarantee that we obtain a balanced tree structure, because it is theoretically possible to construct a QTree that conforms to a list-like structure. For example, if tuples are inserted sorted in a way that the next tuple's coordinates are always higher than those of the previous tuple, then we would add all the data to only one branch of the QTree. However, such an order is unlikely for real data sets and the heuristics we apply keep the QTree almost balanced in practice. Assuming that we have an almost balanced tree, insertion takes only $O(\log b_{max} \cdot f_{max}^2 + c_{src} \cdot \log c_{src})$ time and a lookup $O(\log b_{max} + \log c_{src})$ (Table 5.3).

5.2.3 Construction and Maintenance

So far we have only considered one aspect of constructing data summaries, i.e., how to insert data. We have not yet considered when and what data we need to index. In this respect, we identify two main tasks: i) creating an initial version of a data summary (*initial phase*) and ii) expanding the summary with additional or new information (*expansion phase*).

The construction and maintenance of these data summaries are out of scope for this thesis. However, we give a brief overview of possibilities and sketch the general directions which could be investigated further.

Initial Phase

Once we have constructed an initial version of a data summary, we can use it to determine a set of relevant sources for a given SPARQL query and retrieve relevant documents from the Web. The selection of seed sources to construct the initial version has a strong influence on the ability to discover new and interesting sources in the expansion phase.

Let us assume the case that our data summary covers a subgraph containing only few incoming or outgoing links to the rest of the global Linked Data Web. The lack of links to new sources decreases the probability of further extending the index. Selecting seed sources which provide many links to other documents increases the chance of discovering new sources. The selection of those well interlinked sources can be done via sampling on a random walk over the Linked Data graph or choosing the top ranked sources of existing datasets.

In general, we identify two different approaches for constructing an initial version:

(i) PRE-FETCHING: The most obvious approach is to fetch an initial set of sources to index from the Web using a Web crawler. An advantage of this approach is that existing Web crawling systems can be used to gather the seed URIs. Random walk strategies, in particular, generally lead to representative samples of networks and thus result in a set of sources that could serve as a good starting point to further discover interesting sources [Henzinger et al., 1999]. The quality of query answers depends on the selection of the selected sources and depth/exhaustiveness of the crawl.

(ii) SPARQL QUERIES: Another approach is to use SPARQL queries and collect the sources to index from the answer to the queries. Given a SPARQL query, we can use the LTBQE approach to iteratively fetches the content of the URIs selected from bound variables of the query. However, this would require to have at least one dereferenceable URI (preferable a subject URI) in the SPARQL query as a starting point.

The decision which strategy to select strongly depends on the application scenario and has to be chosen accordingly.

Expansion Phase

After having created an initial version of a data summary, there might still be sources whose data have not yet been indexed. Given a SPARQL query, it is very likely that the initial summary contains information about dereferenceable URIs that are not indexed. In this case, the summary should be updated with these newly discovered URIs to increase the completeness of answer sets for future queries. In this context, we distinguish between pushing and pulling sources:

(i) PUSH OF SOURCES refers to methods involving users or software agents to trigger expansion, which can be done for example via a service similar to the ping services of search engines.

(ii) PULL OF SOURCES does not need any external triggers and can be implemented using lazy fetching during query execution. Lazy fetching refers to the process of dereferencing all new URIs needed to answer a query. The approach is similar to constructing the initial data summary using SPARQL queries.

The latter approach sounds appealing since it elegantly solves the cold-start problem by performing a plain LTBQE approach on the first query and successively

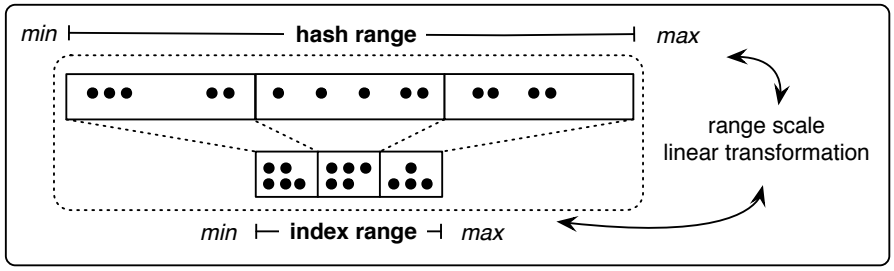


Figure 5.2: Example of range scaling.

expanding the data summary with more relevant sources. Note that the expansion could be combined with pre-fetching for each new query, thus accelerating the expansion of the summary.

5.3 IMPORTANCE OF HASHING

In the following we explain how our system uses hash functions to map RDF statements to numerical values, which are inserted and stored in the buckets of the data summaries. Similar or correlated data items should be clustered and represented in the same bucket. Data summaries adhering to these criteria are ideal for query optimisation and source selection and therefore have a positive influence on precision.

In the case of multidimensional histograms, the hash function should equally distribute the input data among the buckets. The equal distribution is not a requirement for the QTree, since the buckets are adjusted to the input values.

There is a wide variety of hash functions which map strings to integer or float values. A trivial class of the hash functions interpret the encoding of a string as an integer or long value. Another widely used group of hash functions represents the string values with its computed checksum, fingerprint or digests, e.g., one can use the CRC checksum or encryption algorithms like SHA-1. More advanced hash functions try to minimise possible bijective mappings from different strings to the same hash value. Other functions are order preserving, that means that the order of the hash values reflect the order of the input data; e.g., the alphabetical order of the strings.

A common method allowing for efficient aggregation is to normalise the hash values by scaling them from the numerical range of the hash function into a smaller range. One possible way of scaling is to use a linear transformation as depicted in Figure 5.2. The figure also illustrates how a range scale improves clustering and leads to an uniform distribution of the data.

To define the numerical range of a data summary, we have to consider two special cases:

(1) TARGET RANGE IS TOO BIG (SPARSE DATA): If the target region is too big, most of the target range is likely not to be occupied at all. This strongly affects the quality of the multidimensional equi-width histogram whereas the QTree was designed to handle sparse data – see Section 5.2 for more details.

(2) TARGET RANGE IS TOO SMALL: If the target range is too small, we have to deal with hash value collisions, i.e., different strings are mapped onto the same numerical value although their original hash values were different. This will lead to false positive decisions for source selection.

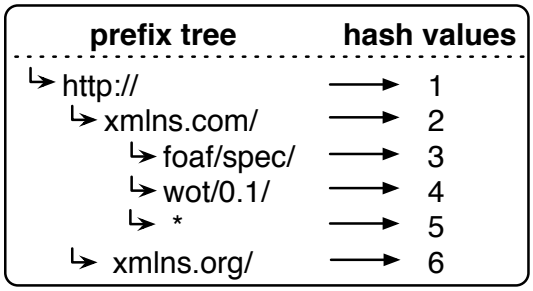


Figure 5.3: Prefix Hashing.

Thus, selecting the target range is a crucial task and directly influences the query processing.

5.3.1 Hash Functions

The general approach is to apply the hash function to the string value of each node in the RDF triple. Alternatively, one can consider applying different hash functions to the different types of RDF terms, namely resources, blank nodes and literals and/or considering the position of the RDF term (subject, predicate or object).

In this thesis, we focus on the following three hashing approaches:

STRING HASHING (STR):

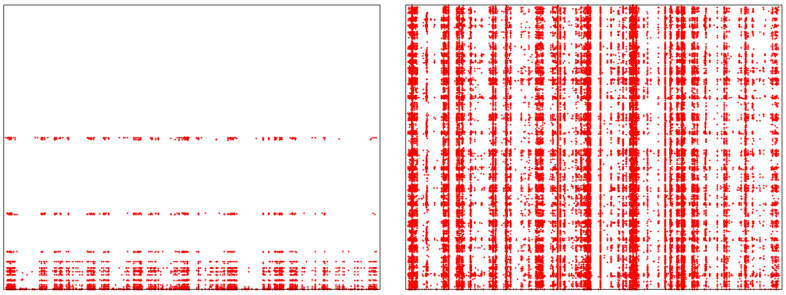
This group of hash functions computes hash values based on checksums and fingerprints. The advantage is that these algorithms aim at providing a unique hash value for each string value and thus try to use the all the available numerical space.

PREFIX HASHING (PRE):

Prefix hash functions use a prefix tree to map a string value to the value of the longest matching prefix in the tree and thus provide a good clustering for similar strings. The basic structure of a prefix tree is depicted in Figure 5.3. In the example we can see that all string values starting with `http://xmlns.com/` are mapped to values between 2 and 5. For example, the URI `http://xmlns.com/foaf/spec/name` is hashed to 3. A string only consisting of the prefix `http://xmlns.com/` is hashed to 5. The advantage of prefix hashing is that it provides a better clustering of similar RDF values compared to the string hashing. However, prefix hashing reduces the number of possible values, especially if the prefix tree does not contain many prefixes. In addition, we have to maintain a prefix tree which can consume a lot of space because of the number of prefixes. However, early experiments showed that a QTree with prefix hashing performs better than a string hashing in terms of the quality of the source selection.

MIXED HASHING (MIX):

The mixed hashing function combines the both presented approaches of prefix and string hashing. Subject and object values are hashed using prefix hashing and predicate values are hashed with string hashing using checksums. The reason for applying different hash functions for the different position of the RDF terms is that earlier experiments revealed that the number of distinct predicates on the Web is rather small (in the number of ten thousand)



(a) Mixed hashing (b) String hashing

Figure 5.4: 2D plots of the hash values of different hashing functions for RDF terms at the predicate (x-axis) and object (y-axis) position.

compared to the possible number of subject and object values. A prefix hash function would map several predicates to the same hash value and we would loose important information. If we apply a string hashing function for the RDF terms at the predicate position we conserve more information because each predicate will be mapped to a unique hash value. This is especially important for the join processing as we will discuss in detail in Section 5.4.

5.3.2 Comparison of hash functions

The distribution of the input data for mixed and string hashing is shown in Figure 5.4 as two-dimensional plots. We omit the prefix hashing from the figure because the distribution patterns are very similar to the mixed hashing, with the only difference that the predicate dimension contains more data points for the mixed hashing. The plots show the distribution of the hashed RDF statements for the RDF terms at the predicate (x-axis) and object (y-axis) position. We selected these two dimensions because they show best the difference between the two hashing approaches and are representative for the other dimensions. The input dataset is a breadth-first Web crawl of RDF/XML documents starting from a well connected source (more information about the dataset in Section 5.5.1). We can see that the string hashing equally distributes the input data over the allocated numerical space. The mixed hashing shows a higher clustering of the input data and leaves large areas of the numerical space empty. Based on these patterns, we can conclude that in theory string hashing should be more suitable for histograms and a prefix or mixed hashing favours the QTree. Our evaluation provide several proofs that the theory holds in practice.

5.4 SOURCE SELECTION

An advantage of the data summaries we advocate here is that they support estimation of relevant sources by processing parts of a query plan before actually fetching any sources. In this section, we provide details on how source estimation works for single triple patterns and joins of triple patterns. Table 5.4 summarises the notation we use throughout the section. Note that for ease of exposition we refer to buckets that contain a single `count` value and a list of sources contributing to this value – we indicate the required modifications for buckets containing a set of (`count`,`source`) pairs where appropriate (cf. Section 5.2).

Symbol	Explanation
$\mathcal{B}, \mathcal{R}, \mathcal{L}$	sets of buckets (regions)
B, R, L	buckets (regions)
$\oplus$	bucket join operation (used in *region joins*)
$\mathcal{J}$	join space (special set of buckets)
$R[i]$	i-th dimension of bucket R
$R[i].hi, R[i].low$	max and min value of R in i-th dimension
c_R	cardinality of bucket R
$\mathcal{S}_R$	set of sources contributing to bucket R
s_r	number of results that source s contributes to
$\lvert\mathcal{R}\rvert$	number of buckets in set $\mathcal{R}$
$\overleftrightarrow{R}^j$	extension in join dimension j of bucket R
$\overline{\mathcal{R}}^j$	average extension in join dimension j of all buckets in $\mathcal{R}$
$\frown$	"overlapping" relation
$\sqcap_j$	overlapping intervals in dimension(s) j of two buckets

Table 5.4: Used symbols.

5.4.1 Triple Pattern Source Selection

Single triple patterns define the leaf operators of query plans, where relevant data is extracted from the Linked Data sources. For determining the sources that can contribute to a join, we first determine the sources that can contribute to these basic triple patterns. With the help of the data summaries, source selection is achieved by determining the buckets (i.e., the data regions) that correspond to a triple pattern. Therefore, a triple pattern is converted into a set of coordinates in numerical space by applying the used hash functions to the elements of the pattern. Triple patterns containing only constants map to a single point in the three-dimensional space, while variables result in spanning the whole range of hash values for the respective dimension, thus constructing a cubic region corresponding to the triple pattern. Intuitively, several filter expressions can be included in the construction of such a *query region*. This includes all filter statements that can be mapped directly to according hash values (e.g., range expressions, but not contains expressions). Algorithm 1 summaries the complete procedure to determine relevant sources for a given triple pattern.

Based on the constructed query region R, we can determine all buckets contained in the data summary that overlap with R. In multidimensional histograms, all buckets are inspected in sequence. In contrast, the hierarchical structure of a QTree supports to start at the root node and then to traverse all child nodes if their minimal bounding boxes (MBBs) overlap R. All buckets on leaf level visited by this tree traversal constitute the set of relevant buckets.

After having identified all relevant buckets, we determine the percentage of overlap with R. Let $\text{size}(R)$ denote the size of a region R, c_B the number of data items (cardinality) represented by bucket B and O the overlapping region of B and R. Then, the cardinality of O is calculated as $c_B \cdot \frac{\text{size}(O)}{\text{size}(B)}$. Based on the overlap, the bucket's source URIs, and the cardinality (i.e., the number of represented RDF

Input: BGP b, QTree QT, min/max dimensional extensions dimSpec
Output: list of relevant buckets (containing sources)

```
1  for i = 0 to 2 do
2  |  if b[i] ≠ variable then
3  |  |  R[i].low = hash(b[i]);
4  |  |  R[i].hi = hash(b[i]);
   |  else
6  |  |  R[i].low = dimSpec[i].low;
7  |  |  R[i].hi = dimSpec[i].hi;
   |  end
   end
10 B = ∅;
11 for B ∈ QT : B overlaps R do
12 |  O = B.overlap(R);
13 |  c_O = c_B · size(O)/size(B)};
14 |  B = B ∪ {(O, c_O, S_B)};
   end
16 return B
```

Algorithm 1: Source selection for triple patterns.

triples) we can determine the set of relevant sources and the expected number of RDF triples per source – assuming that triples are uniformly distributed within each bucket. Thus, the output of the source selection algorithm is a set of buckets, each annotated with information about the overlap with the queried region, source URIs, and the associated cardinality.

5.4.2 Join Source Selection

The presented source selection for triple patterns (and filter statements) already reduces the number of sources that have to be fetched for processing a query. However, we can reduce that number even further if we include the join operators into the pre-processing of a query. The buckets determined for single triple patterns act as input for the join source selection. As it is likely that there are no join partners for data provided by some of the sources relevant for a triple pattern, this will reduce the number of relevant sources. Thus, we consider the overlaps between the sets of obtained relevant buckets for the triple patterns with respect to the defined join dimensions and determine the expected result cardinality of the join. In the general case, a join between two triple patterns is defined by equality in one of the dimensions. Thus, we have to determine the overlap between buckets in the join dimensions, while leaving other dimensions unconstrained.

The performance of processing joins on the data summaries depends on several factors, where the most relevant are:

1. the order of joins
2. the actual processing of the join operation

As in relational databases, the first point should be handled using a cost estimation for different join orders and the second one by choosing between different join implementations. Before we discuss these basic optimisations, we will illustrate the general principle of such *region joins*.

5.4.2.1 *Region Joins*

The crucial question is how we can discard any of the sources relevant for single triple patterns, i.e., identify sources as irrelevant for the join. Unfortunately, if a bucket is overlapped, we cannot omit any of the contributing sources, because we have no information on which sources contribute to which part of the bucket. To not miss any relevant sources, we can only assume all sources from the original bucket to be relevant. Sources can only be discarded if the entire bucket they belong to is discarded, such as the smaller bucket R_2 for the second triple pattern in Figure 5.5. Thus, data summaries and hashing functions that result in small sets of small buckets promise to be particularly beneficial for the join source selection and the overall performance of our query processing approach.

The result of a join evaluation over two triple patterns is a set of three-dimensional buckets. Joining a third triple pattern requires a differentiation between the original dimensions, because the third triple pattern can be joined with any of them. For instance, after a subject-subject join we have to handle two different object dimensions; a join between two three-dimensional overlapping buckets results in one six-dimensional bucket with an MBB that is equivalent to the overlap. In general, a join between n triple patterns results in a $(3 \cdot n)$-dimensional *join space*.

Example 5.4. We exemplify the join source selection algorithm based on Query 5.1:

```
SELECT ?f
WHERE {
  dblpP:HartigBF09 foaf:maker ?a .
  ?x owl:sameAs ?a .
  ?f foaf:knows ?x .
}
```

Query 5.1: Friends of the authors of a paper.

The query consist of three triple patterns. The first two triple patterns are joined over the object position and the last two are a subject-object join. As a remark, executing this query with the LTBQE approach would fail, since the source `dblpPDoc:HartigBF09` contains no `owl:sameAs` information to provide solutions for the second query pattern.

Figure 5.5 illustrates the first step of join source selection on the basis of the introduced example Query 5.1. We assume that the first join is processed over the first and second triple pattern, i.e., an object-object join over `?a`.

The sets of input regions for each triple pattern are determined as described in Section 5.4.1 on the basis of the queried predicate. For simplicity, we assume this results in only one bucket for the first and two buckets for the second triple pattern. Each resulting bucket corresponds to a slice of the three-dimensional space. With regard to the join dimension, there are two overlapping buckets. Both overlapping buckets L_1 and R_1 are constrained by their overlap in the join dimension. Other dimensions are not constrained. Thus, the shaded parts of both buckets represent the result buckets of the join. For a join between different dimensions, e.g., a subject-object join, the approach is the same. The subject dimension of the first triple pattern restricts the object dimension of the second, and vice versa.

Figure 5.6 illustrates how the second join from example Query 5.1 is processed. The join involves `?x1`, i.e., an object-subject join between the second and third triple pattern. For illustration purposes, we omit the predicate dimensions and show equal dimensions on the same axis (slices of the six-dimensional space re-

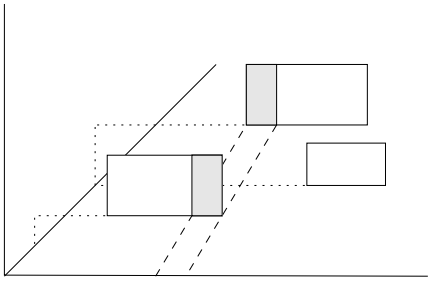


Figure 5.5: Region join between first and second triple pattern.

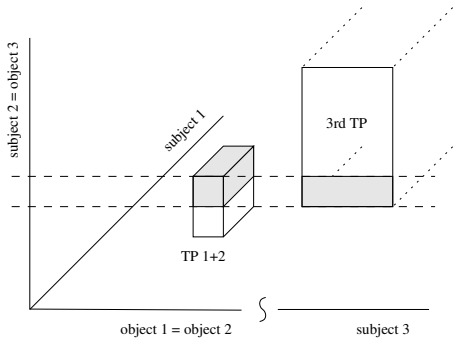


Figure 5.6: Region join with third triple pattern.

duced to the three shown dimensions). The left-most bucket $L_1 \bowtie R_1$ corresponds to the bucket resulting from the first join. One bucket S_1 shown from the result of the third triple pattern overlaps with it, i.e., there is an overlap between the `subject 2` dimension of $L_1 \bowtie R_1$ (originally, the subject from R_1) and the object dimension of S_1. The resulting overlap defines the nine-dimensional result bucket, containing information about all resources that might contribute to this bucket.

5.4.2.2 *Region Join Implementations*

The general principle of determining sources relevant for joined triple patterns can be implemented in several different ways. Basically, the different alternatives known from relational databases can be mapped to the processing of region joins. We discuss the alternatives available in our query engine in the following.

NESTED-LOOP JOIN A straightforward implementation of a region join is a *nested-loop join*.

Algorithm 2 provides a detailed illustration of this implementation. The overlap between the both input sets is determined by two nested loops (lines 2 and 3). Overlapping buckets are joined in the inner loop (line 3) using existing methods for determining the overlap between buckets.

The $(3 \cdot (i + 1))$-dimensional regions resulting from the i-th join are stored in a join space $\mathcal{J}$. This join space acts as one input for the next join. Thus, the join operator actually processes one such join space and the three-dimensional regions for a triple pattern – for the first join, $\mathcal{J}$ corresponds to the three-dimensional buckets resulting from the left-most triple pattern in the join tree. Note that, after the

Input: join space $\mathcal{J}_{in}$ (left-hand side), set of buckets $\mathcal{R}$ (right-hand side), left/right join dimension l/r
Output: new join space (containing relevant buckets and sources)

1 $\mathcal{J}_{out} = \emptyset$;
2 **forall** buckets $L \in \mathcal{J}_{in}$ **do**
3 **forall** buckets $R \in \mathcal{R}$ **do**
4 **if** $\exists O_L = L[l].\text{overlap}(R[r])$ **then**
5 $O_R = R[r].\text{overlap}(L[l])$;
6 $c_{O_R \oplus O_L} = \frac{c_L \cdot \frac{\text{size}(O_L)}{\text{size}(L)} \cdot c_R \cdot \frac{\text{size}(O_R)}{\text{size}(R)}}{\max(L[l].hi - L[l].low, R[r].hi - R[r].low)}$;
7 $\mathcal{J}_{out} = \mathcal{J}_{out} \cup \{O_L \oplus O_R, c_{O_R \oplus O_L}, \mathcal{S}_L \cup \mathcal{S}_R\}$;
  end
  end
end
11 **return** $\mathcal{J}_{out}$

Algorithm 2: Nested-loop join.

first join, two of the six dimensions are equal. Handling them separately is just for ease of understanding and implementation. The $\oplus$ operator in line 7 symbolises the operation of combining two buckets while increasing the number of dimensions accordingly: the three dimensions from O_R are added to the $3 \cdot i$ dimensions of O_L, together forming the $3 \cdot (i+1)$ dimensions of the result bucket. The new cardinality $c_{O_R \oplus O_L}$ (line 6) of the resulting bucket is determined using the percentage of overlap for both buckets (cf. Section 5.4.1), assuming uniform distribution in both buckets. We provide details in Section 5.4.3. The set of relevant sources $\mathcal{S}_{O_R \oplus O_L}$ is a union over the sets from both buckets.

Input: left input $\mathcal{L}$, right input $\mathcal{R}$
Output: join space containing overlapping buckets

```
forall L ∈ ℒ do
    forall R ∈ ℛ do
        𝒥.add(determineOverlap(L, R));
    end
end
return 𝒥
```

Algorithm 3: Principle of nested-loop join.

A simplified description of the nested-loop join principle is depict in Algorithm 3. Note that we omit the restriction to the actual join dimensions. We use this abbreviated form to show the differences to other implementations. The resulting number of operations is $\Theta(|\mathcal{L}| \cdot |\mathcal{R}|)$, i.e., we have to call method `determineOverlap` exactly $|\mathcal{L}| \cdot |\mathcal{R}|$ times.

INDEX JOIN Intuitively, the efficiency of the join processing can be increased using special join indexes. One option is to use an *inverted bucket-index* that stores mappings from the values of a dimension to relevant buckets. We illustrate such an index on the left in Figure 5.7. The references from values to buckets can be used during join processing to efficiently determine all regions that contain a certain value. Note that this is the same principle as in a hash join. However, rather than applying a hash function to the join values, we only have to collect the references from dimension values (which are in fact, hash values) to buckets. For clarification

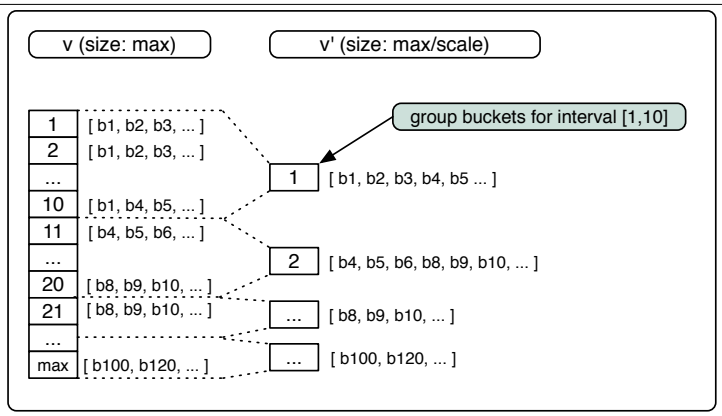


Figure 5.7: Illustration of an inverted bucket index.

and to differentiate from the general problem of hashing for the data summaries we call this join *index join*. A full index of that kind can result in a very high in-memory requirement. In the worst case – when all buckets span over the whole dimension range – for each dimension value we have to store the references to all buckets. To lower the memory requirements, we introduce an approximation by storing references for a range of values rather than single values. This is depicted in Figure 5.7 on the right, assuming that we use a scale factor of 10 (i.e., ranges of size 10: $[1, 10], [11, 20]$).

Ranges that are not covered by any bucket are omitted in the index. In an index without approximation we have to store $\sum_{R \in \mathcal{R}} \overleftrightarrow{R}^j = \overline{\mathcal{R}}^j \cdot |\mathcal{R}|$ entries, where $\mathcal{R}$ refers to the set of regions that have to be indexed, $\overleftrightarrow{R}^j := R[j].hi - R[j].low + 1$ to the range of region R in join dimension j, and $\overline{\mathcal{R}}^j := \frac{1}{|\mathcal{R}|} \cdot \sum_{R \in \mathcal{R}} \overleftrightarrow{R}^j$ to the average range of all buckets from $\mathcal{R}$ in join dimension j. The size of the approximated variant cannot be determined exactly for the general case, as it depends on the actual distribution of regions in the whole range of $\mathcal{R}$. However, we can provide an upper limit as $O(\frac{\overline{\mathcal{R}}^j \cdot |\mathcal{R}|}{scale})$, where $scale$ refers to the used scale factor.

Input: left input $\mathcal{L}$, right input $\mathcal{R}$
Output: join space containing overlapping buckets

```
idx = buildInvertedBucketIndex(R);
forall L ∈ L do
    O = idx.getOverlappingBuckets(L);
    forall O ∈ O do
        J.add(determineOverlap(O, L));
    end
end
return J
```

Algorithm 4: Principle of index join.

Algorithm 4 illustrates the principle of the index join. As known from hash joins, we choose the smaller of both input sets to build the inverted bucket-index. In the algorithm, without loss of generality, we assume that this is $\mathcal{R}$. The method `determineOverlap` has to be called for all pairs of overlapping regions, as we need the fraction of overlap to estimate the number of results for join ordering (Section 5.4.2.3) and source ranking (Section 5.4.3). The exact overlap has to be determined only once per overlapping pair of regions, no matter how much they

actually overlap. This is assured by method `getOverlappingBuckets`, which returns all overlapping buckets at once using the before created index.

A lower limit of the number of operations using the index join is provided by $\Omega(|\mathcal{R}| + |\mathcal{L}|)$, while the worst-case upper limit is

$$O(|\mathcal{R}| + |\mathcal{L}| \cdot |\mathcal{R}|). \tag{5.1}$$

The exact number depends on the actual distribution of buckets in $\mathcal{R}$ and $\mathcal{L}$ and can be approximated by

$$\begin{aligned} \Theta \quad (\quad & |\mathcal{R}| + |\mathcal{L}| - |\{L \in \mathcal{L} : L[j] \frown \mathcal{L} \sqcap_j \mathcal{R}\}| \\ & + \sum_{L \in \mathcal{L} : L[j] \frown \mathcal{L} \sqcap_j \mathcal{R}} |\{R \in \mathcal{R} : R \frown L\}|), \end{aligned} \tag{5.2}$$

where $\frown$ refers to the overlap relation and $\mathcal{L} \sqcap_j \mathcal{R}$ to the range(s) where buckets from $\mathcal{L}$ and $\mathcal{R}$ overlap in join dimension j. First, we need to scan all $|\mathcal{R}|$ buckets to build the index, then we have to scan all $|\mathcal{L}|$ buckets from $\mathcal{L}$. Only for those buckets from $\mathcal{L}$ that are in the overlap with $\mathcal{R}$ (i.e., all $L \in L : L[j] \frown \mathcal{L} \sqcap_j \mathcal{R}$), we have to actually call `determineOverlap` for each overlapping $R \in \mathcal{R}$. Thus, the index join becomes more efficient with smaller sizes of $\overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j$ and smaller average bucket sizes $\overline{\mathcal{L}}^j$ and $\overline{\mathcal{R}}^j$. To make this more evident, we can provide an asymptotic upper limit assuming uniform distribution of the buckets in $\mathcal{L}$ and $\mathcal{R}$ and an unapproximated inverted bucket-index. Then, the *density* of a set of buckets $\mathcal{L}$ can be determined as

$$d_{\mathcal{L}} := \frac{|\mathcal{L}| \cdot \overline{\mathcal{L}}^j}{\overleftrightarrow{\mathcal{L}}^j}.$$

The density describes the fraction of points in the whole range $\overleftrightarrow{\mathcal{L}}^j$ of $\mathcal{L}$ that are covered by buckets from $\mathcal{L}$. Using the density of a set of buckets, we can approximate the upper limit for the sum from Equation 5.2 as

$$\begin{aligned} & O\Bigg(\underbrace{\frac{d_{\mathcal{L}} \cdot \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j}{\min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}}_{|\{L \in \mathcal{L} : L \in \mathcal{L} \sqcap_j \mathcal{R}\}|} \cdot \underbrace{\frac{d_{\mathcal{R}} \cdot \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}{\min(\overline{\mathcal{R}}^j, \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j))}}_{|\{R \in \mathcal{R} : R \text{ overlaps with } L\}|} \Bigg) \\ = \; & O\Bigg(\frac{\frac{|\mathcal{L}| \cdot \overline{\mathcal{L}}^j}{\overleftrightarrow{\mathcal{L}}^j} \cdot \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j}{\min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)} \cdot \frac{\frac{|\mathcal{R}| \cdot \overline{\mathcal{R}}^j}{\overleftrightarrow{\mathcal{R}}^j} \cdot \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}{\min(\overline{\mathcal{R}}^j, \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j))} \Bigg) \\ = \; & O\Bigg(|\mathcal{L}| \cdot \underbrace{\frac{\max(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}{\overleftrightarrow{\mathcal{L}}^j}}_{\leqslant 1} \\ & \cdot \quad |\mathcal{R}| \cdot \underbrace{\frac{\max(\overline{\mathcal{R}}^j, \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j))}{\overleftrightarrow{\mathcal{R}}^j}}_{\leqslant 1} \Bigg). \end{aligned}$$

This shows that $|\mathcal{L}| \cdot |\mathcal{R}|$ is a very rough upper limit in Equation 5.1. In fact, the amount of operations reduces significantly with decreasing $\overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j$, $\overline{\mathcal{L}}^j$ and $\overline{\mathcal{R}}^j$.

5.4.2.3 *Join Ordering*

Besides the actual join implementation, the second crucial aspect for achieving a good performance in join processing is the join order. In principle, other well-known optimisation techniques can be mapped directly to the problem of ordering

region joins. We just have to use an appropriate cost model. The crucial cost factor

Input: set J of pairs $\{\mathcal{L}, \mathcal{R}\}$ of triple pattern results to join
Output: list containing ordered joins

```
O = ⊥; min = ∞;
while J ≠ ∅ do
    forall {L, R} ∈ J do
        if |L| · |R| < min then
            min = |L| · |R|;
            next = {L, R};
        end
    end
    O.add(next);
    min = ∞;
    J = J \ {next};
end
return O
```

Algorithm 5: Greedy algorithm for join ordering.

for region joins is the number of resulting buckets. Thus, for the time being, we implemented a greedy algorithm using a cost function that estimates the number of resulting join buckets, as shown in Algorithm 5. The input for the algorithm is a set of pairs of relevant buckets for triple patterns that can be joined, as determined by the triple pattern source selection. Based on the number of buckets that might result from a join between two triple patterns in the worst case, the algorithm chooses the cheapest join in a greedy manner. One could extend this simple cost model by statistics describing the distribution of buckets in order to estimate the actual number of comparisons for each join.

The optimisations discussed above are only of basic nature. The focus of this work lies on the general applicability of hash-based data summaries for querying Linked Data. As part of that, we also analyse the general benefit we gain from optimising region joins (Section 5.5.2), which is expected to form only a small part of the entire query processing overhead.

5.4.3 Result Cardinality Estimation and Source Ranking

As source selection is approximate, the set of relevant sources will usually be overestimated, i.e., contain false positives. Please note that false negatives are impossible: any region where results exist are guaranteed to be covered by the buckets of the summaries. Moreover, some queries may actually be answered by a large set of sources, such that a focus on the most relevant ones becomes important. Both issues suggest to introduce a ranking for sources identified as being relevant for answering the query.

One approach to rank sources according to their relevance is to use the cardinalities provided by the data summary. The intuition is that sources that provide many results should be ranked higher than sources providing only a few results. Thus, the idea is to estimate the number of results s_r that each source $s \in \mathcal{S}$ contributes to. The ranks are assigned to sources according to the values of s_r in descending order.

If each QTree bucket B provides an estimated cardinality c_B and a list of associated sources $\mathcal{S}_B$, we could simply assume uniform distribution and assign $c_B/|\mathcal{S}_B|$ to each source of a bucket, while summing up over all buckets. In early tests we recognised that this ranks sources very inaccurately. A simple modification of the summaries, which results in constant space overhead, is to record the cardinality c_B^s for each source contributing to a bucket separately. More specifically, c_B^s es-

timates the number of results in B that source s contributes to, summed over all joined triples. Thus, $c_B = (\sum_{s \in S_B} c_B^s)/jl_B$, where jl_B represents the join level of B (i.e., the number of triple patterns that have been joined to form one data item in B). This helps to overcome the assumption of a uniform distribution in the bucket. The number of results a source contributes to is determined as:

$$s_r = \sum_B c_B^s$$

Line 6 in Algorithm 2 can be adapted by applying the formula separately for each source, while substituting c_B by c_B^s, c_L by c_L^s and c_R by c_R^s.

To provide an example, we assume that bucket L_1 from the first triple pattern in Figure 5.5 summarises 60 triples from a source s_1 and 40 triples from a source s_2. Further, bucket R_1 from the second triple pattern shall refer to 20 triples from source s_2 and 50 triples from source s_3. The ratio between overlap and bucket is $\frac{2}{7}$ for L_1, respectively $\frac{1}{4}$ for R_1, and the larger bucket R_1 has an extension of 40 in the object dimension. Thus, after the first join we rank the sources as follows:

1. s_3 : contributes to $s_1 \bowtie s_3$ and to $s_2 \bowtie s_3$: $\frac{60 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} + \frac{40 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} = 8.93$ of the join results
2. s_2 : contributes to $s_1 \bowtie s_2$ and $s_2 \bowtie s_3$, and doubled to $s_2 \bowtie s_2$: $\frac{60 \cdot \frac{2}{7} \cdot 20 \cdot \frac{1}{4}}{40} + \frac{40 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} + 2 \cdot \frac{40 \cdot \frac{2}{7} \cdot 20 \cdot \frac{1}{4}}{40} = 8.57$ of the join results
3. s_1 : contributes to $s_1 \bowtie s_2$ and to $s_1 \bowtie s_3$: $\frac{60 \cdot \frac{2}{7} \cdot 20 \cdot \frac{1}{4}}{40} + \frac{60 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} = 7.5$ of the join results

Note that we estimate the *contribution* of each source to the join result. Thus, for each pair of joined sources we count twice – one time for the left-hand side, one time for the right-hand side. The estimated cardinality for the join result is actually half of the sum over all sources, i.e., 12.5 in the example. The determined cardinalities for each source are stored in the resulting bucket $L_1 \bowtie R_1$. They are used in the same way for result cardinality estimation and source ranking after the second join, which is still a rough approximation but can already significantly improve query processing performance. In order to guarantee that we do not miss any relevant source, we cannot discard any of the sources, no matter how small the estimated contribution is. Remember that the uniform distribution is just an assumption to enable a cardinality estimation at all. The source ranking based on this helps to assess the importance of all relevant sources. The effect is grounded in probability laws, by which the probability that a source contributes to a fraction of a bucket (the region resulting from the join overlap) increases with its total number of data items in the bucket.

Due to the assumptions we make during ranking, sources providing a large number of triples will usually be ranked higher than smaller sources although both large and small sources can potentially contribute to the query result. However, as we show in Section 5.5.2, source ranking based on cardinality works satisfyingly accurate for real-world sources.

There are several possible approaches to improve the ranking accuracy, e.g., by inspecting the importance for each join dimension separately and determining a combined rank in the end. A crucial question that has to be answered before is: What should be the target of the ranking? In our approach, we rank sources higher that likely contribute to many results of the join. Alternative approaches can be based on the popularity of the sources using, for instance, PageRank [Brin and Page, 1998; Page et al., 1999], HITS [Kleinberg, 1999], and optionally external information from Web search engines. Another alternative is to directly rank the importance of the generated join results rather than the importance of the sources

that contribute to them. Ranking in our approach is very important and represents an orthogonal research problem in itself.In Section 5.5.2 we show that the current approach already indicates the actual importance ranking of sources in a satisfyingly accurate manner.

5.5 EVALUATION

5.5.1 Setup

Our experiments aim at providing insight into memory consumption, runtime efficiency and query completeness for various lightweight index structures for source selection. We describe the setup of our experiments, methods and the test data in the following and discuss the obtained results afterwards.

We benchmark core functions of the index structures:

INDEX BUILD: The index structures should be able to handle the insertion of RDF data streamed directly from a crawler. We measure the time and memory needed to insert a given number of statements. We expect that our results are very close to the theoretical complexity analyses.

QUERY TIME: Ideally, the source selection of the index structure returns only relevant sources for conjunctive SPARQL queries in milliseconds. The experiment executes SPARQL queries with two implemented join operators – the nested loop and inverted bucket list operator – and with and without the join ordering optimisation. We measure the execution time for the different operators and the QTree and multidimensional histogram.

SOURCE SELECTION: Hash-based data summaries, SLI and II return an estimation of sources relevant to answering a query. Due to the approximate nature of these indexes the returned sources are always a superset of the actually relevant sources. Our experiment is designed to measure the average number of estimated sources in comparison to the average number of actually relevant sources. We expect that the number of selected sources is higher than the number of actually relevant sources.

QUERY ANSWERING: The query results should justify the expensive execution of a SPARQL query directly over live Web content (we highlighted the challenges that are involved in fetching the content for the selected sources; e.g politeness and also temporarily server problems). Considering a source ranking, a Linked Data query processor would ideally use only the top-k sources to assure a worst case query time and still guarantee a certain degree of completeness of the results. We evaluates the quality of the source selection and the completeness of the query answers wrt. the top-k selected sources. The baseline results are derived from the test queries over the materialised data set.

5.5.1.1 *Datasets and Queries*

We use two real-world datasets collected from the Web with the LDSpider [Isele et al., 2010] framework. Two data set were gathered with a breadth-first crawl starting from a well connected RDF source with a content filter for sources that contain `application/rdf+xml` files. This gathering method is similar to the BTC dataset (which we presented in Section 3.1). For this experiment we required a smaller subset of the Linked Data Web which is also connected. As such, we performed our

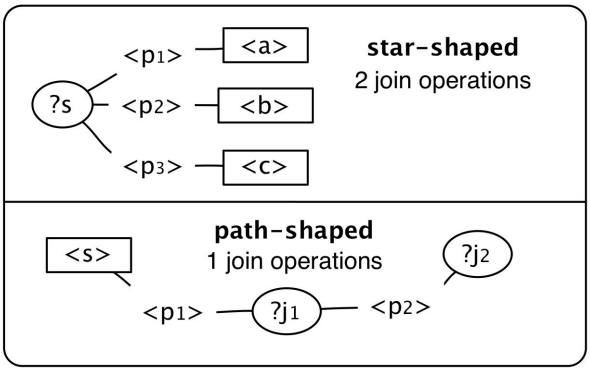


Figure 5.8: Abstract illustration of used query classes.

own crawl rather than extracting a subset from the BTC dataset, which, we believe, would be more challenging than a straightforward crawl. As a result, we obtained a large dataset L consisting of 3.1m RDF statements from 15.7k sources and a smaller data set M which contains the first 50% of the content of L.

We used our QWalk query generator, as presented in Section 4.4.2, and experimented with randomly generated queries corresponding to two general classes The first class of queries consist of *"star-shaped queries"* with one variable at the subject position. The second type of queries are *"path-shaped queries"* with join variables at subject and object positions. Figure 5.8 shows abstract representations of these query classes. The query classes of choice are generally understood to be representative for real-world use cases and are also used to evaluate other RDF query systems (e.g., [Neumann and Weikum, 2010]).

The star-shaped queries were generated by randomly picking a subject URI from the input data and arbitrarily selecting distinct outgoing links. Then, we substituted the subject in each BGP with a variable. Path queries were generated using our random walk approach. We randomly chose a subject URI and performed a random walk of pre-defined depth. The result of such a random walk was transformed into a path-shaped join by replacing the connecting nodes with variables.

Using these approaches, we generated from the data 50 queries for each query class with zero, one and two join operations. We use `P-`n to denote path queries with n join operations and `S-`n to denote star-shaped queries with n join operations.

5.5.1.2 *Setup*

We use a dual core AMD Opteron 250 server with 4GB of memory and two 1TB hard disks, running Ubuntu 9.10/x86_64 (latest stable release of Ubuntu) for our experiments. We base our experiments on implementations in Java, and use Version 1.6 of the Sun Virtual Machine.

We used reference implementations for the schema-level index and the inverted URI index:

- Schema-level Index (**SLI**): We use the standard Java HashMap implementation to store the list of source URIs that contain a property p and/or the object (class) of an `rdf:type` statement. We arrive at two maps: one has as keys the RDF properties and as values a list of sources containing the corresponding property, and the second map has as keys the object of type statements and

again as values the list of sources containing the corresponding type statements. To select relevant sources, we extract all RDF properties and classes contained in the query, perform lookups on the two maps, and return the union of the results of the lookups as relevant sources. Note, this does not involve any kind of ranking.

- INVERTED URI INDEX (**II**): Our reference implementation of an inverted URI index also uses in-memory Java HashMaps, where the keys are the URIs in the dataset and the values are lists of source URIs which contain the key URI. We select relevant sources for a given query by extracting all URIs contained in the query, perform for each URI a lookup in the map and return the union of all lookup results. Note, this does not involve any kind of ranking.

The reason why we use reference implementation are the following: Our focus is not on the actual index and query performance, rather we are interested in how suitable these indexes are for the source selection and thus focus only on number of the resulting sources relevant to a query. The number of estimated sources is the crucial performance factor of our proposed query system as we will show in the evaluation section.

5.5.2 Results

In the following we present and discuss the results of our experiments. An overview of the concrete setup of our evaluation is given in Table 5.5. We provide for each experiments a selection of the results in plots, and compare and discuss the complete results.

5.5.2.1 *Index Build*

The time needed for inserting a certain number of statements for MDH and QT is shown in Figure 5.9. The plot shows (with log-scaled xy-axis) the time elapsed to insert n-statements (`total` labelled point lines) for MDH and QT with different parameters.

We can see that MDH (two bottom lines) performs by order of magnitudes better than QT. The best insert time of MDH was achieved with string hashing with an average of 45 statements per millisecond. Among different QT configurations, mixed hashing performs best and needs in average 30 ms to insert a statement (or 0.04 statements per millisecond). Further, we observe that MDH increases the ratio of the inserted statements per time (cold start), whereas QT is getting slower. The differences between the insert times are due to the speed of the hash function in the case of MDH. The string hash functions are in general faster than the hash functions based on the prefix tree. In contrast, the string hashing functions significantly slow down QT. In general, string hashing equally distributes the string hashes over the allocated numerical space, which causes the QTree to recompute the bucket boundaries for nearly each inserted statements. Prefix based hashing clusters the input data more and reduces the operations to optimise the bucket boundaries in the QTree. (despite the fact that the computation time of the prefix hashing is slower than the string hashing). We can observe in all insert experiments that the at a certain stage we reach linear insert times (this corresponds with the theoretical analysis of insert complexity).

A complete summary of the results of the insert benchmark is presented in Table 5.6 (including the results for our reference implementation of II and SLI). The first column contains the index types and hash function as described in Table 5.5. The value in the brackets indicate which data set was used as an input. The column *compression* shows the fraction of the index size compared to the size of the

Datasets	
L	#stmt: 3.1m, #src: 16k
M	#stmt: 1.53m, #src: 6.6k
Index Types	
QT_L	QTree(b_{max}: 50k m_{max}: 1m f_{max}:20)
QT_S	QTree(b_{max}: 25k m_{max}: 1m f_{max}:20)
MDH	Histogram(b_{max}: 50k m_{max}: 1m)
SLI	Schema-level index
II	Inverted URI index
Hashing	
MIX	Prefix and String hashing
STR	String hashing
Queries	
S-i, $i = 0, 1, 2$	50 Star-shaped query with i join(s)
P-i, $i = 0, 1, 2$	50 Path-shaped query with i join(s)
Query Planning	
Op_N	Nested-loop join operator
Op_I	Inverted bucket index join operator

Table 5.5: Overview of the setup of our experiments.

raw data (561M for dataset L and 260M for dataset M). The table shows again the relatively poor indexing performance of QT. However, we will show that QT outperforms the other approaches in the quality of the source selection at the price of indexing time. Please not that the current implementation is a proof of concept, implemented with the standard Java data structures. We did not implement any low-level optimisations such as bit-based index structures or string encoding for the stored source URIs. For instance, we store full names of sources in each bucket together with the cardinality values. The general advantages with respect to the space complexity of the hash-based data summaries proposed in this work, i.e., scaling with the number of sources but not with the number of triples, are discussed in Section 5.2.2.

Index version	Index Size (MBytes)	Compression	Index Time (sec)	avg. stmts/ms
QT_L-MIX (L)	42	7.4%	65147	0.04
QT_L-STR (L)	54	9.6%	112835	0.02
QT_S-MIX (L)	30	5.3%	48763	0.06
QT_L-MIX (M)	32	5.7%	45345	0.04
MDH-STR (L)	36	6.4%	67	45.5
MDH-MIX (L)	11	1.9%	122	24.40
SLI (L)	12	2.13%	49	62.35
II (L)	49	8.7%	56	54.30

Table 5.6: Index size and insert time of different approaches.

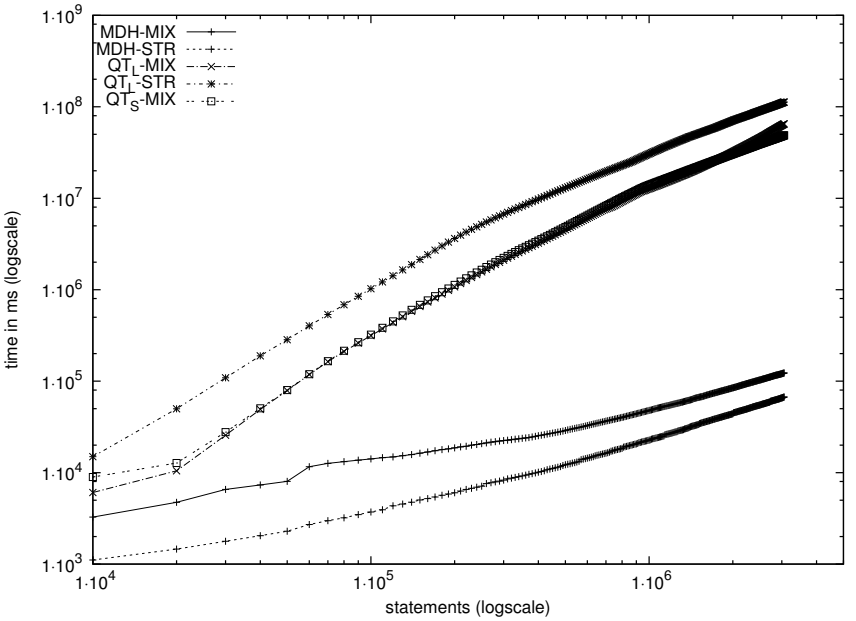


Figure 5.9: Insert time per statements.

5.5.2.2 *Query Time*

We first discuss the results of the join ordering optimisation and the various join operators (Figure 5.10) and second, we present total query times for different setups of the QTree in Figure 5.11 and Figure 5.12.

As we highlighted already (cf. Section 5.4.2.3), we integrated a basic join ordering method which optimises the query processing. The results in Figure 5.10 show that join ordering has a huge impact for the runtime of the source selection. The values above the plot boxes show the time benefit of the ordering for the query execution. The values are relative to the query time without ordering, thus a negative value indicates that the query processing with join ordering is faster by the amount of the value; e.g path-shaped queries with two joins are 59% faster using join ordering and an inverted bucket list operator.

The plots show the average time needed to execute a query without and with the join ordering optimisation for the nested loop (Op_N) and inverted bucket list (Op_I) join operator. The selected index is QT_L with hashing functions MIX and STR. We omit in the plot the query runtime for other QTree or histogram versions for a better readability. However, we observed the following effects also for the other index versions. The big plot boxes show the query time without the ordering and the smaller light boxes the query time with ordering. The study of the results shows that join ordering achieves a large runtime benefit for path-shaped queries, whereas we cannot really see an optimisation effect for star-shaped queries. The difference is mainly due to the type of queries. Star-shaped queries contain constants at the predicate and object position which results in a more selective list of buckets, whereas our path-shaped queries have only one triple pattern with two constants and the other contain only one specified constant which makes join reordering more beneficial. We save more than 60% of the query time for path shaped queries with two joins for the QTree and over 50% for the star shaped queries with two joins.

Next, we present complete query runtime for different index versions, query types, join operators and our reference implementation of SLI and II. As expected,

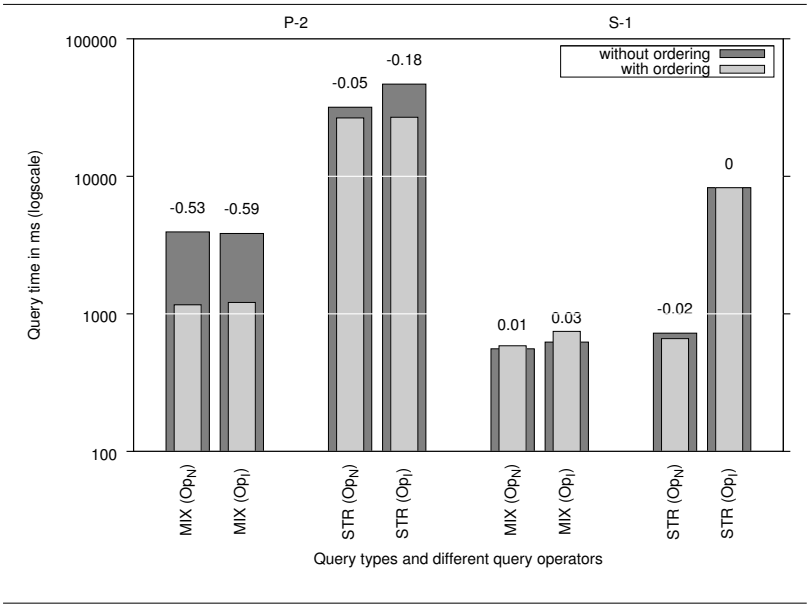


Figure 5.10: Average query time with/out join ordering.

the reference implementations provide nearly constant query times outperforming the data summary indexes. The plots are limited on the y-axis to 10 seconds for better readability of the graphs. In addition, average query times of more than 10 seconds are not very applicable in a real world scenario. The query times of the reference implementation are for all query classes less than 300 ms for SLI and less than 100ms for II. We can see in Figure 5.11 that the average query time is less than 1 second for all query types using a nested loop join operator (beside the two outliers for path-shaped queries with two join variables and an index using a string hash function). Moreover, we can see that MDH is slightly faster than the QT variants. The expected benefit by applying a inverted bucket index join operator was not observed (Figure 5.12). We can see that this version of a join operators has query times for many of the query types of more than 10 seconds. Further, we observe an influence of the hashing function used; string hashing slows down query processing of QT and speeds up query processing of MDH.

In summary, we can say that the nested loop join operator outperforms the inverted bucket list join operator. II offer the fastest (and nearly constant) query times followed by SLI and MDH. QT shows the slowest performance in most of the queries.

However, these results about the actual source selection times are only one operation in the entire query processing and not the crucial factor. The most expensive part in evaluating a query is the dereferencing of the relevant sources via HTTP GET. Considering all the issues with accessing Linked Data (as highlighted in Section 5.1) we can expect that the number of estimated sources is the crucial factor.

5.5.2.3 *Source Selection*

An important aspect is the quality and amount of the relevant sources estimated by the index structures. Figure 5.13 shows two interesting characteristics of the different index structures and query types and can be interpreted as follows: The difference between the number of estimated and real sources is the number of sources which are falsely selected as relevant. We show only the results for one setup of an QTree (QT_LMIX) with mixed hashing and a histogram with string hashing MDHSTR. These two setups return the best query answer completeness and thus,

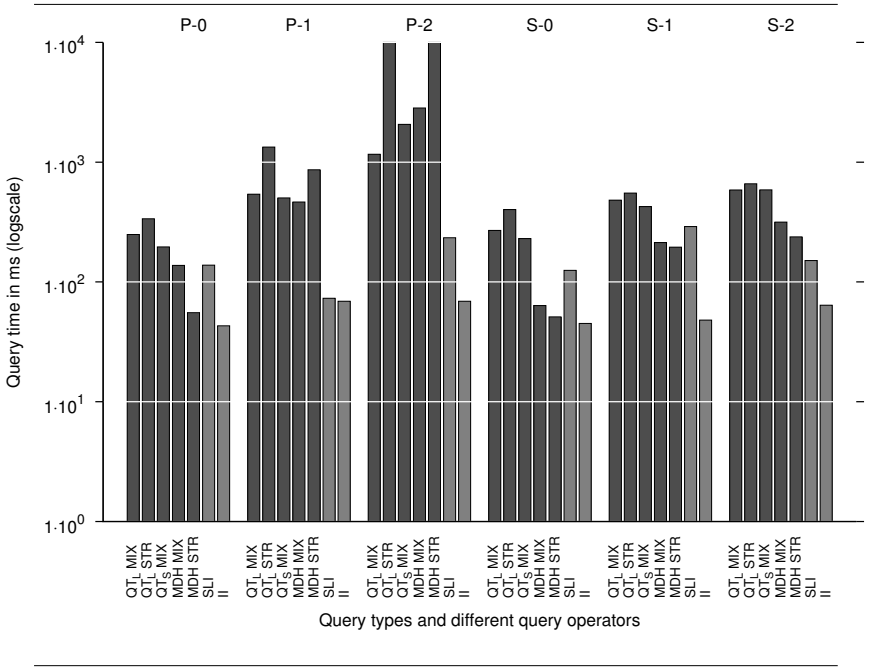


Figure 5.11: Query time for different query types and approaches using a nested loop join operator.

we decided to use them as the best representatives for the source selection evaluation. First, we see that the number of real query relevant sources are increasing with the number of joins for path-shaped queries and again, decreasing with the number of joins for the star-shaped queries. The same effect can be observed for the number of estimated sources for the path-shaped queries. As a surprise, the estimated number of sources relevant to answer the star-shaped queries is also increasing with the number of joins, which stands in contrast to the number of real query-relevant sources. Second, we find that QT shows the best performance in terms of the number of estimated sources to answer a given query. QT estimates the least number of sources for all types of queries. SLI, in contrast, estimates for nearly all query types the largest amount of sources.

Next we discuss the results of the query time to estimate the relevant sources and the number of the estimated sources itself. Let us consider an average lookup time of 500 ms to dereference a relevant source on moderate hardware that allows to perform 75 lookups in parallel (cf. [Hogan et al., 2011]). Further, let us assume that all the returned sources are equally distributed over at least 75 domains which would allow us to perform the lookups in parallel. Thus, a round of 75 lookups would take in average 500 ms. From the results in Figure 5.13 we can see that QT outperforms the other approaches in the total number of estimated sources (for some queries by a factor of ten). The number of estimated sources for QT ranges between 600 and 1200, which would require 8 and 16 lookup rounds or a fetching time of 4 to 8 seconds. MDH, SLI, and II estimate for all query types at least 1500 sources. Fetching the sources requires at least 20 lookup rounds or 10 seconds. The difference in the number of estimated sources slows down the query processing by at least 2 seconds. If we consider the query time to estimate the relevant sources and the time needed to fetch the content of the sources we can conclude that QT clearly outperforms the other approaches.

Eventually, the estimated time and performance of the source lookup operation is idealised and in a real-world setup one cannot assume that we can fetch all

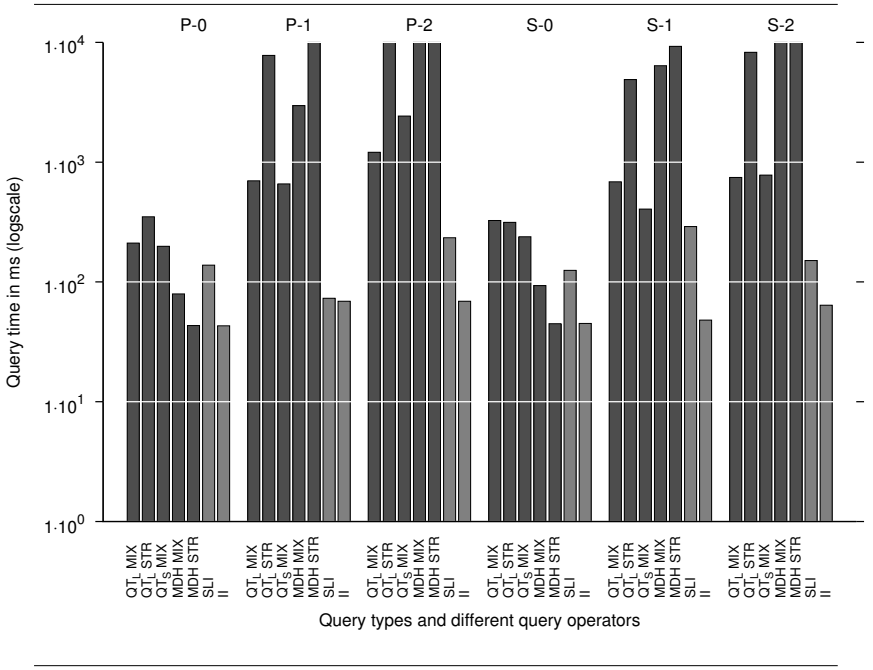


Figure 5.12: Query time for different query types and approaches using a inverted bucket list join operator.

relevant sources in parallel while being polite at the same time. Thus, to be able to reduce the number of relevant sources we implemented basic ranking functions and evaluated these ranking functions next. Please note that the ranking also decreases the effects of false positives.

5.5.2.4 *Source Ranking*

Querying all sources that were determined as relevant will still be too expensive in most cases for two reasons: (i) a huge number of sources may actually contribute to answering a query, where some of them contribute only minimally; and (ii) due to the approximation of the data summaries there may be false positives. For both issues, ranking becomes crucial. Ideally, false positives and sources that contribute only few results are ranked lower than the sources that are actually relevant and can contribute the majority of results. Then, we can still provide a satisfying degree of completeness by querying only the top-k sources. The study of the top-k results presented in Figure 5.14 shows that QT significantly outperforms MDH. There are missing values for the histogram with mixed hashing for path-shaped queries with 2 joins (P-2) due to the fact that these queries timed out in our experiment (query timeout was set to 3 minutes). As our reference implementations for SLI and II do not support any kind of ranking and the returned sources are in random order, we omit the presentation of the results.

The source selection in QT returns over 40% of a query answer with the top-200 sources independent of the query type. MDH achieves only a query answer completeness of maximum 20% with the top-200 sources for all query types. We can observe that QT returns with the top-200 sources on average over 80% of the result statements for simple lookups with either the subject (S-0) or object (P-0) defined as a variable. For queries with one join (P-1,S-1) we still get over 60% of the results (with the top-200 sources). Moreover, our simple ranking algorithm yields 40% of the results with only the top-10 sources for simple lookups and queries with one join.

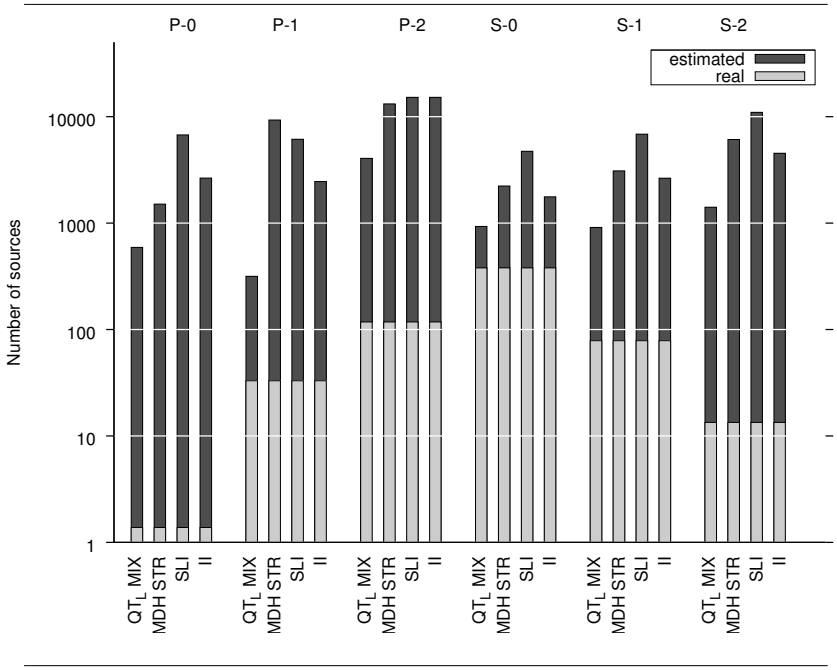


Figure 5.13: Number of estimates and real query relevant sources (the dark area of the bars are the false positives).

We can observe the same correlation between the hashing function and the index structure as in our other experiments. A string hashing favours the histogram and returns more results than a histogram with the mixed hashing. In contrast, the QTree performs better with the mixed hashing function than with a string hashing. In addition, we can see that the QTree version with more buckets performs better than with less buckets for the same input data.

Furthermore, Figure 5.14 illustrates the influence of b_{max} on our approach. The general rule is that the more buckets we allow a QTree to use (higher b_{max}), the less approximation has to be used in order to meet the b_{max} constraint. Therefore, the QTree provides more details in terms of smaller buckets and the number of false positives is reduced. However, by increasing b_{max}, we also increase the required space in main memory. So, we need to find a tradeoff between false positives and index size. In general, the influence of f_{max} is negligible. As b_{max} and f_{max} are QTree specific parameters, discussing their influences in detail is out of the scope of this thesis.

We decided not to perform the actual live lookup of the query processor since there are many factors involved which we cannot influence or ignore. Some of the main factors are the time-dependent usage of the network and the available bandwidth which affect the download rate of the source contents. Another factor is the domain distribution of the selected sources which impacts politeness scheduling. In the best case we have k distinct domains in the top-k selected sources and can fetch the content with k parallel lookups. In the worst case we have a single domain in the top-k sources and need to perform k sequential lookups with a wait time between each request. We refer also to our discussion and findings about the server reliability and repeatability of the experiments presented previously in Chapter 4.

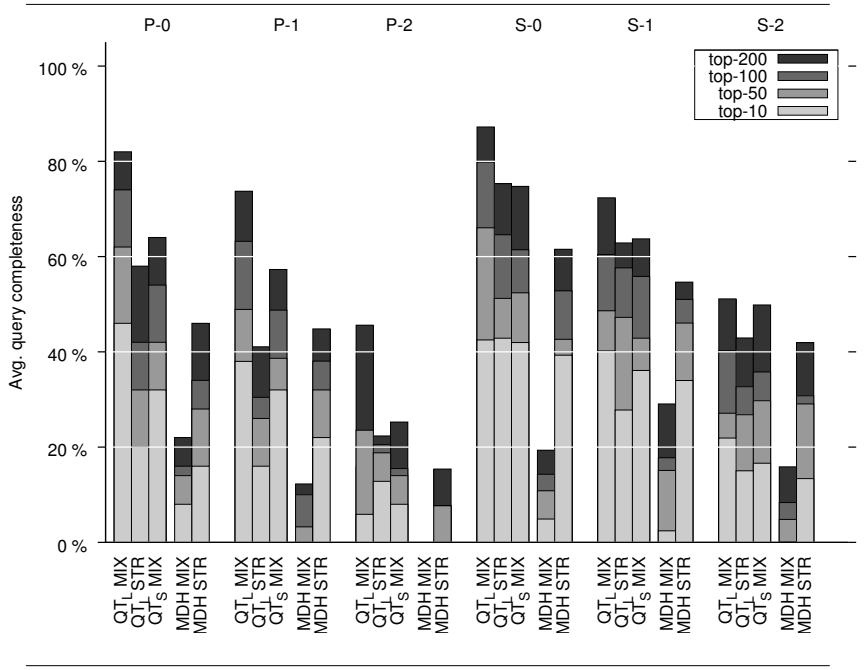


Figure 5.14: Average query completeness.

5.5.3 Discussion

Based on our experimental results, we can state that the data summary approach, instantiated here by the QTree, represents a promising alternative for querying Linked Data on the Web.

The quality of the source selection and the fraction of possible answers is reasonably good using the QTree and rather poor for the other approaches. The QTree data structure outperforms the other approaches with the least number of estimated sources and the best query completeness. The basic query planning algorithms, together with the straightforward ranking algorithm, provide reasonably good query times and answer completeness for simple lookups and queries with one join operation. The QTree achieves a query answering completeness of over 40% (compared to global knowledge) with the top-200 sources throughout all our experiments (top value of 90% for star-shaped queries).

Regarding indexing times, the QTree data structure is the slowest approach compared to the alternatives. The difference in performance is due to the fact that MDH uses fixed and predefined bucket boundaries, whereas the QTree dynamically adjusts and optimises the boundaries based on the state of the inserted data. Our reference implementation of SLI and II require only to parse the necessary statements in the input file to build an index.

The QTree is not designed to support bulk indexing of large amounts of data. However, the current insert times are reasonable in a very dynamic setup, in which an agent would index the content of Web sources either during runtime or only from a small number of sources at a time.

In summary, we can clearly state that our approach – using data summaries like the QTree – provides the highest quality for the selection of query-relevant sources. However, as a price for the benefits, the aimed applications have to bear slower index times compared to other approaches, due to the higher complexity of the QTree as a summary. A crucial advantage of the proposed data summaries is that their size grows only with the number of inserted sources, and not with the

number of inserted statements as it holds for the schema-level and inverted URI index.

5.6 CONCLUSION

We presented in this chapter our approach for determining relevant sources for SPARQL queries over RDF published as Linked Data. As these queries are issued ad-hoc, optimisation has to be done for all possible queries and cannot specialise on a specific type or a subset of queries. The presented approach uses hash functions to transform RDF statements into numerical space and data summaries to efficiently describe the data (RDF triples) provided by the sources.

Herein, we presented two variants of data summaries in conjunction with several hash functions and how to construct them. Furthermore, we discussed how to use these summaries to determine relevant sources for queries with and without joins. To limit query execution costs, i.e., the number of queried sources, we proposed the optional use of ranking to prioritise sources. In addition to theoretical analyses, we provided an extensive evaluation highlighting the influence of data summaries, hash functions, and query types on performance. Overall, our results show that our approach is able to handle more expressive queries and return more complete results to queries compared to previous approaches.

An ideal application scenario for the proposed data summary approach should have the following requirements: Firstly, it is not necessary to have complete answers, rather the application will return only the top-k results. Secondly, the focus is on guaranteed up-to-date answers instead of possibly outdated results from old snapshots. Finally, the application allows a certain amount of time to execute and evaluate a given query.

Based on our obtained results, we can see our proposed approach very suitable as the underlying infrastructure for smart Linked Data browsers. A combination of a sophisticated user interface and our QTree approach would allow users to navigate and browse the Linked Data Web. In general, user interfaces over Linked Data use already conjunctive SPARQL queries for the user interactions. Further, these user interfaces do not require to show the complete set of answers for a given query in general; typically, they display only the top-k results (similar to the front-ends of the traditional Web search engines). The current systems use materialised indexes as the underlying index structure which requires a significant amount of on-disk storage capacity and/or do not allow to use complex SPARQL queries. This envisioned solution can be a very interesting alternative to the existing ones and will provide a lightweight application which still offers fast query times and reasonable query completeness over the top-ranked sources. Using live query evaluation over sources which check access control in a decentralised manner would allow for application of Linked Data in corporate environments.

6 HYBRID SPARQL QUERY PROCESSING: FRESH VS. FAST RESULTS

503 SERVICE TEMPORARILY UNAVAILABLE

— Public SPARQL endpoint(s), *2012*

In the previous two chapters we studied Linked Data query approaches which access query relevant data remotely from Web resources at runtime and thus can return up-to-date results. However, the retrieval of remote content from diverse sources at query-time naturally implies much slower response times compared to optimised local indexes which replicate data from parts of the Web. We depict in Figure 6.1 this inherent trade-off between approaches that give *fresh results* versus approaches that give *fast results*, represented at two ends by live query techniques and centralised "*store*" respectively.

Herein, we propose a novel *hybrid query framework* that returns fresher results from a broader range of sources vs. the centralised scenario, while speeding up results vs. the live scenario. Our engine features a original query planner that decides which parts of the query to run live and which to run locally based on knowledge of the *coherence* for triple patterns of the SPARQL store with respect to remote data. The coherence involves both the dynamicity of remote data and the coverage of the store. By getting the store to quickly service query-patterns for which it has good up-to-date coverage, and by running the rest of the query live, our hybrid approach aims to strike a balance between fresh and fast results.

The remainder of this chapter is organised as follows:

- In Section 6.1, we introduce our hybrid query architecture.
- Section 6.2 describes how to collect the necessary coherent estimates for the query planner.
- In Section 6.3, we present details about our hybrid query-planning component, including different reordering and split strategies.

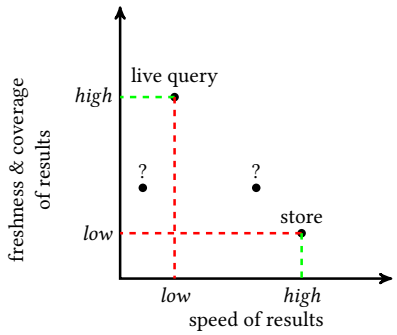


Figure 6.1: Query Trade-off

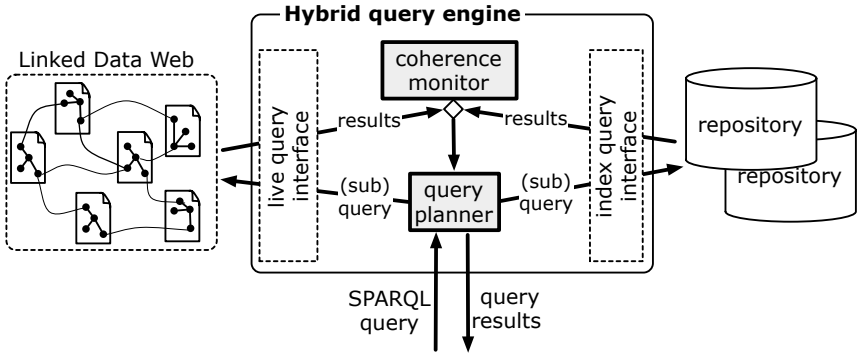


Figure 6.2: Architecture of a hybrid query engine.

- Section 6.4 describes our evaluation methodology to compare different hybrid query plans and presents the results which show the feasibility of the hybrid query execution approach.
- In Section 6.5, we conclude with a critical discussion.

6.1 ARCHITECTURE OF A HYBRID QUERY ENGINE

Our hybrid engine can be thought of as a "live-wrapper" for centralised SPARQL stores that splits a query into two groups, where the patterns in one group are executed over a centralised store and the other part is executed using existing live-querying techniques. Our proposed hybrid query engine has the following targets:

Ⓣ1 *fast response times*, ideally close to those of centralised queries;

Ⓣ2 *coherence of results* close to those of live query processing such that we retrieve fresh answers;

Ⓣ3 *system independence*, i.e., being compatible with any SPARQL engine or live-query processor; and

Ⓣ4 *lightweight implementation* with low resource requirements, particularly regarding main memory.

As we discussed at the beginning, Ⓣ1 and Ⓣ2 are antagonist targets and our query engines tries to find the best trade-off between query time and coherence of the results.

The resulting architecture is illustrated in Figure 6.2 and consist of four core components. The main component in the architecture is the QUERY PLANNER which tries to find an overall "optimal" trade-off for a given request, deciding what parts of the query to delegate to the local and remote engines. With regards to Ⓣ3, we can initialise our architecture with an INDEX QUERY INTERFACE and a LIVE QUERY INTERFACE as black-box components; both consume SPARQL queries and produce SPARQL results, but the former interfaces with a local index, whereas the latter interfaces with the Web. Finally, to help find that trade-off, the COHERENCE MONITOR collects *high-level* empirical statistics (see Section 6.2) about the endpoint's coverage of data for different query patterns compared with the Web. These compact estimates are easy to maintain and their storage costs are low (as per Ⓣ4).

Along these lines, the hybrid query engine offers a SPARQL interface that sits on top of a centralised SPARQL engine and a live query processor. These two interfaces build the connections to the black boxes of SPARQL store(s) and live query processor. The INDEX QUERY INTERFACE can be a (possibly remote) public SPARQL store or any data warehousing approach which offers the SPARQL protocol (e.g., an intranet database). The LIVE QUERY INTERFACE also accepts SPARQL queries and could be based on, for instance, a bottom-up link-traversal engine Chapter 4 or a top-down source selection index (e.g., as we proposed in Chapter 5), or some combination thereof. Here we instantiate the live query processor with our bottom-up, link-traversal based query execution approach (LTBQE). In such scenarios, LTBQE is required to deal with simple sub-queries, which we have shown to be feasible in the previous Chapter 4.

The COHERENCE MONITOR collects information about the coverage and freshness of different triple patterns and sources. The coherence estimates of individual patterns is used by the QUERY PLANNER component to split a given query into two sub-queries—a local and a live sub-query. Eventually, the query processor forwards the local part to the index query interface and the live part is processed over the relevant Web sources *in situ*. We see this conceptually straightforward architecture as a first step towards freshening up centralised results: topics such as adaptive coherence estimates and more fine-grained interaction between the local and remote query processors are left to future work.

Note that since the local SPARQL engine is treated as a black-box (as per (T3)), we cannot influence the design of the physical plan for the static sub-query: we delegate generating the final sub-query plan to the engine, which we assume implements, e.g., local selectivity estimates to organise optimal execution. In the general case, a similar situation exists for the live query processor.

In the following sections, we elaborate further on the COHERENCE MONITOR component (Section 6.2) and the QUERY PLANNER component (Section 6.3).

6.2 COHERENCE ESTIMATION

Given the scope [Bizer et al., 2011] and dynamicity of Linked Data, one can expect that results returned by a centralised store are inherently limited by its coverage of the Web and by the freshness of its local index which our experiment in Chapter 3 showed.

The COHERENCE MONITOR component of our framework computes and stores the coherence estimates of query patterns for an store with respect to live query execution. These coherence estimates are used in the query planner to identify which patterns are more likely to be up to date on the store and which patterns are likely to be stale or missing.

In order to acquire the coherence estimates of a given store, we use the data from our index freshness study experiment, presented in Section 3.3. Briefly recapturing the experiment, we probed both the store and the Web with a broad range of simple SPARQL queries, compared the results and characterised parts of the cache's index that are likely to be stale or missing. Table 6.1, reintroduces the notation from the experiment which we also will use in the following.

We use the results of each query from the experiment to create coherence estimates for our query planner by identifying data predicate–source combinations that are likely to be stale. Again, we view results as consisting of variable–binding pairs (i.e., $\mathsf{S}, \mathsf{L} \subset \mathbf{V} \times \mathbf{UL}$ reusing common notation for the set of all query variables, URIs and literals resp.) and we exclude answers involving blank nodes to avoid is-

S	set of results returned from the store
L	set of results returned by LTBQE
$\Delta^S := S \setminus L$	set of results returned by the cache but not LTBQE
$\Delta^L := L \setminus S$	answers found by LTBQE but not returned by the cache
$\Delta^\cap := L \cap S$	set of *coherent* results

Table 6.1: Notations used for coherence estimation.

sues of scoping and inconsistent labelling. We identified two possible methods by which queries can be used to test coherence.

SOURCE-BASED ESTIMATES Assuming the SPARQL cache uses *Named Graphs* to track the original source of information on the Web, we can compare the data for a Web source against the data cached in the corresponding graph using `GRAPH` queries. However, (and as is the case for the two caches we test later) many stores do not have consistent naming of graphs: sometimes the graph may indeed refer to a particular Web source, but oftentimes the graph will be a high-level URI (e.g., `http://dbpedia.org`), informally indicating a dump from which the data were loaded but which cannot be directly retrieved.

TRIPLE PATTERN ESTIMATES Thus, we instead focus on triple-based estimates. We centre our notion of coherence for triple patterns around predicates. This restricts our approach to triple patterns with a constant as predicate; other patterns are assigned a default estimate. To generate such triple-pattern estimates, we use the results from our experiment. To quantify the coherence of predicates based on the results, we study two measures. To present these, we apply the notations from Figure 3.7 – and which are listed in Table 6.1 – to the results of the probe queries (see Query 3.1), adding subscripts to indicate results for a certain query, e.g., Δ^L_q. We denote results involving a predicate p as, e.g., $\Delta^L_q(p) := \{r \in \Delta^L_q : (\texttt{?pIn}, p) \in r \vee (\texttt{?pOut}, p) \in r\}$, and say $p \in \Delta^L_q$ iff $\Delta^L_q(p) \neq \emptyset$.

QUERY-BASED COHERENCE: The coherence of a predicate p is measured as the ratio of queries for which p appeared in Δ^L. For the full set of queries $\mathcal{Q}$, let $M_q(p)$ denote for how many queries a live result involving the predicate was missing at least once in the cache results ($M_q(p) = |\{q \in \mathcal{Q} : p \in \Delta^L_q\}|$). In addition, we count $L_q(p) = |\{q \in \mathcal{Q} : p \in L_q\}|$ as the queries for which the live engine returned at least one result containing p. The *query-based coherence* of the predicate p is then computed as:

$$\mathsf{coh}_q(p) = 1 - \frac{M_q(p)}{L_q(p)} \quad .$$

RESULT-BASED COHERENCE: For this measure, we inspect the ratio of missing results for a predicate p, rather than the fraction of stale queries. Let $M_r(p)$ denote the count of all live results involving the predicate p that were missed by the cache, summated across all queries ($M_r(p) = \sum_{q \in \mathcal{Q}} |\Delta^L_q(p)|$). Let $L_r(p)$ denote the count of all results involving p retrieved by the live engine ($L_r(p) = \sum_{q \in \mathcal{Q}} |L_q(p)|$). The *result-based coherence* is then:

$$\mathsf{coh}_r(p) = 1 - \frac{M_r(p)}{L_r(p)} \quad .$$

We performed the experiment in early March 2012 and gathered coherence information for 2,550 predicates for OpenLink and 1,627 predicates for Sindice endpoint.

For the two stores under analysis, Figure 6.3a illustrates the number of predicates that fall into different intervals of coherence values; the y-axis is in logarithmic scale, and the linear x-axis intervals represent the coherence measures as percentages (the right of the graph indicates increasingly coherent predicates).

The figure shows that the OpenLink store is more in sync with current Web data than Sindice; we believe that OpenLink was extensively updated in Feb. 2012. We measured that 67% of the tested predicates in the OpenLink index are entirely up-to-date ($coh_r(p) = 1$), versus 30% of the predicates for the Sindice store. In contrast, information for 14% of the tested predicates in the OpenLink index are entirely missing or out-of-date ($coh_r(p) = 0$), versus 40% for Sindice; these high percentages are due to partial coverage of Web sources, outdated data-dumps in the index, and predicates with dynamic values.

№	OpenLink		Sindice	
	pred.	*queries*	*pred.*	*queries*
1	`swivt:creationDate`	510	`swivt:creationDate`	118
2	`vitro:mostSpecificType`	104	`skos:narrower`	48
3	`swivt:wikiPageModificationDate`	45	`skos:historyNote`	43
4	`aims:hasDateCreated`	42	`bibo:doi`	34
5	`madsrdf:hasCloseExternalAuthority`	31	`prism21:doi`	34

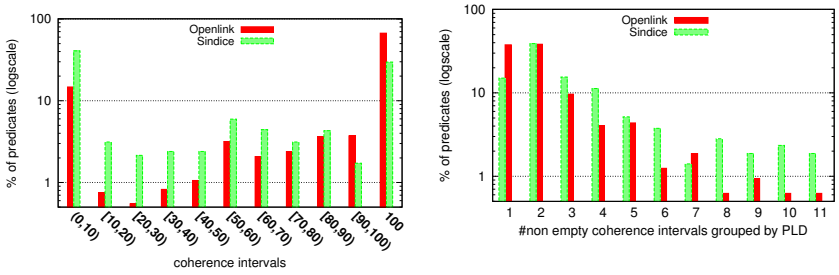
Table 6.2: Most dynamic and prevalent predicates

In more detail, Table 6.2 shows the top 5 predicates where $coh_r(p) = 0$ for both stores, ordered by the number of queries in which they featured as a result.[1] First, we see a mix of dynamic time-stamp predicates that change for every access or modification of a document (`swivt:creationDate`, `swivt:wikiPageModificationDate` and `aims:hasDateCreated`). Second, we see predicates not covered by the index. For Sindice, the incoherent `*:doi` predicates are due to a lack of coverage of the `dx.doi.org` domain and the high incoherency of `skos:` predicates is due to bulk changes in the `esd-toolkit.eu`, `esd.org.uk` and `bio2rdf.org` domains; for OpenLink, the incoherency of `vitro:mostSpecificType` relates to data on the `cornell.edu` domain.

We further analysed the correlation for coherence estimates of the same predicates *across* the two stores. We used Kendall's τ which measures the agreement in ordering for two measures in a range of $[-1, 1]$, where -1 indicates perfectly inverted ordering and 1 indicates the exact same ordering. The τ-score across the two stores was 0.16, with a negligible p-value, indicating a weak, significant and positive correlation between the coherence of predicates for the two stores. The low correlation highlights the store-specific nature of these measures, which are as much about index coverage than about the dynamicity of values. As such, our approach tackles both the global problem of dynamicity and the local problem of index coverage.

Finally, we looked at the correlation between the selectivity of predicates (i.e., how often they occur) and their coherence, which may lead to potential consequences for query planning. Specifically, for each store, we compared the number of (Web) results generated for each predicate across all queries and their $coh_r(p)$ value. The τ-value for OpenLink was 0.1, indicating that less selective patterns tend to have slightly lower coherence; the analogous τ-value for Sindice was -0.03, indicating a *very* slight correlation in the opposite direction. Though limited, we take

1 See `http://vitro.mannlib.cornell.edu/ns/vitro/0.7#` for `vitro:`, `http://prefix.cc/` for others.



(a) Coherence distribution

(b) PLD coherence variations

Figure 6.3: Distribution of predicate coherence values and variation across PLDs

this as anecdotal evidence to suggest that correlation between the selectivity and coherence of predicates is weak, if any.

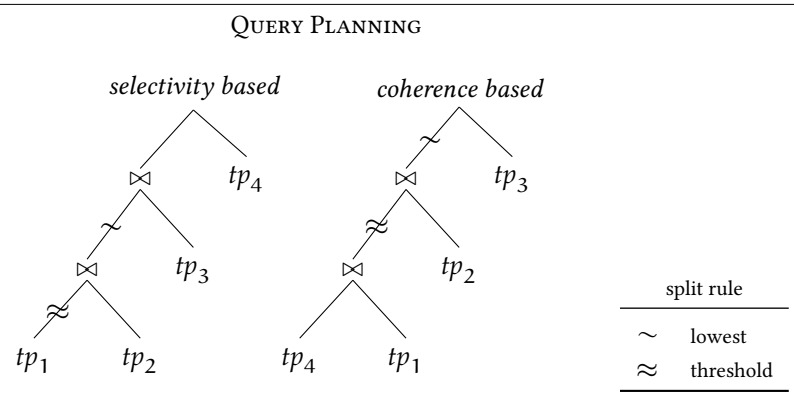
Above, we naïvely assume a single coherence value for predicates in all cases, ignoring subject or object URIs: keeping information for each subject/object would have a high overhead. However, we can generalise subject/object values into pay-level-domains (PLD)[2] and then track coherence for predicate–domain pairs. Thus, we mapped the entity URIs of the queries to their PLDs (581 PLDs with a maximum of 74 queries per domain) and resolved the coherence of predicates for individual PLDs. Focussing on the coherence measure, we divided the scores into eleven intervals as per the x-axis of Figure 6.3b, and for each predicate, count how many intervals it falls into for different PLDs. We observe that the subject and object URIs can be ignored for roughly 40% of the Openlink and roughly 15% for the Sindice predicates. However, we see the importance of tracking coherence for predicate–domain pairs for the remaining predicates. The plurality of predicates (~40%) show two intervals of coherence values.

MAINTENANCE Assuming the cooperation of the SPARQL endpoint, various methods can be used to learn about content changes or updates to the centralised index (similar in principle to internal SPARQL caching proposals as presented by Williams and Weaver [2011]). Data providers may push change notifications to the stores and/or the stores can learn about changes by actively monitoring remote sources as we performed in Chapter 3; this information can then be pushed to the coherence monitor. In a strict black box scenario, where only the public SPARQL interface is available for the cache, one has to periodically re-run or update the gathered statistics, where queries that are observed to return static results could be probed less frequently in an adaptive monitoring setup as proposed by Käfer et al. [2012]. In addition, the system could perform a demand-based or "lazy" maintenance of statistics that is based on, e.g., (i) keeping only frequent query patterns up-to-date or (ii) actively updating coherence estimates as hybrid queries are processed.

6.3 QUERY PLANNER

Our hybrid engine combines local and live query execution to obtain a balance between fresh and fast results. The QUERY PLANNER is responsible for splitting the query into a part for local execution and a part for live execution. Traditional query planning within closed systems focuses on optimising for performance by ordering the execution of query operators to minimise intermediate results. Such query

2 A pay-level-domain is the domain name one has to register and pay for.



	BGP (*s*,*p*,*o*)	$\mathrm{coh}_\tau(\mathrm{p})$	selectivity
tp_1:	`<http://status.samnoble.org/user/1> foaf:knows ?o1 .`	0.86	0.98
tp_2:	`?o1 foaf:interest ?o2 .`	0.32	0.43
tp_3:	`?o1 swivt:wikiPageModificationDate ?o3 .`	0.00	0.21
tp_4:	`?o2 foaf:accountProfilePage ?o0 .`	0.91	0.15

Figure 6.4: Example hybrid query plans for different orderings and splits

planning often relies on *selectivity estimates*, which indicate the amount of results a given operation will generate.

For hybrid query planning, we wish to optimise for both speed *and* freshness. Thus, analogous to (and in combination with) selectivity estimates that optimise for speed, we need other metrics to optimise for freshness. Recalling that (sub-)queries will often be answered faster by materialised indexes than live engines, in the interest of speed, we wish to push as much of the query processing to the store. However, we only want to send requests for which the cache has fresh data available. Hence, along with selectivity estimates, we also need *coherence estimates* to measure how well synchronised the cache is wrt. the Web.

Example 6.1. At this stage, we introduce a simple and abstract example to better explain our query planning approach which is described in the following in detail. Figure 6.4 depicts two different hybrid query plans (selectivity bon the top) for a query consisting of four triple patterns (at the bottom part of the figure). The left query plan shows the query execution if the triple patterns are ordered by their selectivity. Whereas the right query plan shows the execution order based on a coherence based ordering. The coherence and selecitivy values for each triple pattern is also listed in the figure. Moreover, the figure also contains some possible split choices for the different query plans.

6.3.1 Split Types

The core aim of the query planner is to decide which parts of the input query should go to the cache, which parts should go live, and how the obtained results can be combined. The crucial question question for the query planner is: How to split a query?. There exists two high-level options:

MULTI-SPLIT

In this approach, there is no restriction placed on how many splits are made in the query or on which parts go where. This is the most general case. However, if the query is split into many parts, processing the query will involve a lot of coordination of interim results and synchronisation between the cache and the live engines.[3] Thus, this then requires a lot of coordination and synchronisation between the engines. Also, if the cache is accessed through a public interface, it may not allow a sufficient query rate for this approach to work.

SINGLE-SPLIT

Another simpler (but more restricted) option is to perform a *single-split* and separate the query into two queries: a cache and a live sub-query. Much less coordination is then required between engines. Assuming nested evaluation, an open question is whether the cache or live request should be run first.[4]

Considering the practicability of the different split types, we argue that a single split of the query plan into one local and one live part is more practical in the open Web environment. While in theory it would also be possible to use multiple splits, the resulting intertwined dependencies between the local and live parts would lead to very complex query planning, and would require shipping bindings back and forth between the live and local engines. Second, and most important in our scenario, this will result in several small queries sent to the store in high frequency, which is usually blocked by public accessible SPARQL stores.[5] Thus, advanced splitting approaches are better suited to controlled environments (i.e., not public SPARQL stores).

EXECUTION ORDER Given the focus on a split into (at most) two parts, the results of the first executed sub-query serve as input bindings for the second part. We must then decide whether the local part is processed first (i.e., at the bottom of the query plan) or last (i.e., at the top of the plan). We argue to first execute the bottom part of the query against the local cache and afterwards the top part of the query live against the data on the Web. This order of execution has the benefit that:

(i) it only require a single query to be run against the cache's materialised index, thus avoiding overloading the public interface;

(iii) it does not pose any restriction on the triple patterns in the local part, thus avoiding constraints such as the requirement for dereferenceable URIs in some of the query patterns for the live query execution of SPARQL queries (as shown in Chapter 4 and Chapter 5);

(iv) running the local part first, does not only obtain the results from the store more quickly, but also provide additional dereferenceable URI bindings for the live querying phase (passed to the live query using the `VALUES`, previous `BINDINGS`, clause in SPARQL 1.1).

Next, we must then decide the execution order of join patterns given that we decided to use only one split and execute the bottom part first against the store,

3 Where supported, SPARQL 1.1 `VALUES` could be helpful to ship bindings, but would still require a synchronisation point for each split.

4 If both parts of the query are deemed to have low selectivity, they could be run in parallel and hash-joined.

5 This could be solved using SPARQL 1.1 `BINDINGS`, but Virtuoso has no support.

6.3.2 Reordering Strategies

We can see in our example the emphasis for the two different query execution orders. An intuitive approach, which we call *coherence-based ordering*, is to build a query plan with the most coherent patterns at the bottom for local execution, and the most incoherent patterns at the top for live execution. This increases the likelihood that the final result set is fresh and it limits the number of patterns executed live. However, the most coherent patterns may also be the least *selective* (i.e., return the most bindings) thus inflating the number of intermediate results to process. Consequently, this can increase the number of bindings for the patterns executed live, potentially hurting the performance. Because of this and backed by the absence of correlation between coherence and selectivity (cf.Section 6.2), we also consider another approach following traditional *selectivity-based reordering*, where the most restrictive patterns are executed first reducing intermediate results.

COHERENCE-BASED ORDERING

> Query patterns are ordered by the coherence of cache data available for them; coherence gauges the coverage and freshness of cache indexes for answering a pattern (more coherence implies broader and fresher cache coverage). Coherence measures can be computed for patterns based on analysis of the dynamicity of Web data, using probe queries against the cache which can be compared with live results, or (assuming cooperation of the cache) by listening for updates to the cache indexes [Williams and Weaver, 2011]. A single-split strategy is appropriate for this ordering, where patterns that can be best answered by the cache will be executed first and the rest then answered live.

SELECTIVITY-BASED ORDERING

> Query patterns in the join tree are first ordered by selectivity. Selectivities for each pattern can be computed by rule-based estimates or variable-counting techniques [Schwarte et al., 2011]), from analysis of prevalence of patterns in Web data, by sampling data from the cache using probe queries, or (where supported) by posing SPARQL 1.1 `COUNT` queries or queries for statistical summaries against the cache indexes. Patterns that match and return fewer data are executed earlier to minimise interim results.

Both orderings have inherent advantages and disadvantages. The coherence-based ordering maximises freshness but makes no guarantees about the size of interim results. This ordering lends itself well to a single-split strategy, where all of the highly-coherent patterns can be first sent directly to the cache; for deciding where to split, a threshold can be used as mentioned before, or a simple fixed-position split strategy can be employed (e.g., always send only the least coherent pattern live). However, in this approach, the cache sub-query may have a low selectivity and require the cache to materialise a lot of results. In the best case, high selectivity and high coherence correlate with each other such that there is no conflict in the fresh vs. fast trade-off. However, as we will see for experiments in the next section, this is not necessarily the case.The selectivity-based ordering should minimise interim results but makes no guarantees about freshness. In particular, (in)coherent patterns will be mixed throughout the query tree.

The remaining question for the query planner is to: WHERE TO SPLIT A QUERY?.

6.3.3 Split Pattern

The term *split pattern* refers to the position in the query plan in which the query is divided. Everything below the split is executed locally, and everything above and

including the split pattern is executed live. We identified three different options to find the split position for a given query

MOST INCOHERENT PATTERN

Following the same intuition of executing low-coherence patterns live, one option is to choose the *most incoherent* triple pattern as the split. However, the store may still receive highly incoherent patterns (below the max) for which it will return incoherent results.

THRESHOLD BASED

Another approach is to define a constant value indicating a *threshold of incoherence*, where the lowest pattern breaching the threshold becomes the split pattern; this ensures that the store does not receive patterns that are highly incoherent.

FIXED SPLIT POSITION

A further option is to split by a *fixed position* n, whereby the n bottom patterns are executed by the store and the rest are run live. Choosing between the different split options affects the core trade-off of fresh vs. fast results, and thus may depend on individual user needs.

6.3.4 Expected Query Performance

Eventually, given our example in Figure 6.4, we see in the selectivity-based ordering that different types of coherence thresholds may lead to more patterns being run live than when explicitly ordered by coherence. Conversely, at the base of the plan for the coherence-based ordering, we see that tp_4 will return a lot of intermediate bindings (since it has a low selectivity) and does not share a join variable with tp_1 on the right-hand side of the join. In general, one may expect that the selectivity-based operator order would provide low answer times by minimising intermediate bindings, but would return less fresh results since low coherence patterns can appear below the split. However, this ordering also ends up pushing more patterns live since patterns with low selectivity and high coherence are often above the split. Conversely, a coherence-based ordering will lead to more intermediate results, but will run more patterns locally. Thus, a general conclusion about which ordering is preferable is not possible; we instead compare combinations of orderings and splits on an empirical basis in Section 6.4.

In fact, since we consider the SPARQL store and the live-query component as black boxes, the local and live parts of the query will be reordered by the respective engines, thus mitigating some of the performance penalty associated with the possibly naïve ordering used to decide the split in the hybrid query plan. For example, referring back to the coherence-ordered plan of Figure 6.4, if the lowest coherence split rule is applied, the store may internally decide to run tp_1, tp_2 and then tp_4 in that order, avoiding the (huge) expense of running tp_4 first.

Finally, we highlight that the hybrid query planner is responsible for ordering, splitting, delegating and executing sub-queries. The sub-queries sent to the cache or to the live engines are then subject to internal optimisations. This is particularly relevant for single-split coherence-ordering, where the sub-query delegated to the SPARQL interface of the store will be reordered according to internal selectivities. We focus on evaluating conjunctive queries (i.e., SPARQL BGPs) which LTBQE supports. Other features of SPARQL, except `OPTIONAL` (and `MINUS` & [`NOT`] `EXISTS` in SPARQL 1.1) can be layered on top.

6.4 EVALUATION

We now present the setup and methodology to evaluate our proposed hybrid query execution and critically discuss our results.

6.4.1 Setup

Our concrete goals for this evaluation can be summarised as follows:

(i) *prove of concept* and show that with the correct plan, hybrid query execution can extend and freshen up local results while speeding up live results;

(ii) to *evaluate different query plan strategies* by comparing

(iia) selectivity- and coherence-based ordering and

(iib) different split strategies for the query planning.

In parallel, we are interested to see how useful our coherence estimates are for the hybrid-query planning phase. Crucially, we wish to evaluate our proposals in a realistic setting. To do so, our evaluation is run against the two selected public accessible stores: Sindice and OpenLink.

To be able to evaluate the above mentioned points, we need a large and diverse set of queries which helps us to achieve a good overview of how our approach performs in a realistic scenario.

6.4.1.1 *Evaluation Queries*

We require a set of evaluation queries that are answerable by a Linked Data query engine. We would like these queries to have broad coverage of diverse Web sources in order to properly test coherence estimates and hybrid splits. Hence, we used again our QWalk query generator, presented in Section 4.4.2, to generate queries from the Billion Triple Challenge 2011 dataset, which covers a broad range of Web documents as shown in Section 3.1.

We apply our random walk technique on the dataset to select random paths between dereferencable URIs in the data. The resulting queries guarantee to return non-empty results if executed with the linked-traversal based query execution (LTBQE) LIVE QUERY INTERFACE (see Chapter 4 for more details).

Using this method, we produce 200 SPARQL `SELECT` queries of different shapes (star, path, mixed), with varying numbers of patterns (2–6). We randomly assigned distinguished variables to each of the generated queries. Further, we ensure that each query contains at least one pattern above and below a coherence threshold of 0.5. This guarantees that our queries are suitable for a hybrid execution.

6.4.1.2 *Selectivity-based Query Planning*

In practice, we create our hybrid SPARQL query plan using ARQ based on a "*variable counting*" technique [Stocker and Seaborne, 2007] for the selectivity based ordering. This method estimates the selectivity of different triple patterns based on rules involving the number and position of variables it contains. The basic function of this ordering heuristic is that execution costs increase with the number of variables in the triple pattern and but also considering the position of the variables. A variable at the subject position has also a higher execution cost as a variable at the predicate or object triple position. In more detail, we list all possible combination of variables in triple patterns by descending order of estimated execution costs (variables are denoted with "?", query constants with #): `?s ?p ?o .` > `?s ?p #o .` > `?s #p ?o .` > `#s ?p ?o .` > `?s #p #o .` > `#s ?p #o .` > `#s #p ?o .`

notation	*description*
`sel-*`	selectivity-based ordering
`coh-*`	selectivity-based ordering
`*-best`	best theoretical split
`*-incoh`	split at the most incoherent pattern
`*-thr`	split based on a coherence threshold of 0.5
`*-rnd`	random split (i.e., by guessing)
`*-1`	query is split after the first pattern
`*-2`	query is split after the second pattern
`live`	live query approach
`ep`	SPARQL store

Table 6.3: Overview of notation used in the evaluation.

Noteworthy, the variable counting technique could be replaced with cost-based planning using empirical selectivity estimates. However, we would need to obtain the statistics about the underlying data, either again by probing the store or with published statistics. Ideally, every SPARQL store would publish such statistics as voiD files [Alexander and Hausenblas, 2009]. However, following a rule-based approach is more in line with targets Ⓣ3 system independence, and Ⓣ4 lightweight implementation (cf. Section 6.1). A coherence-based operator order is supported by reordering the triple patterns in the query plan produced by ARQ based on their coherence values.

6.4.1.3 *Execution*

To evaluate different orders and different cut-off positions, we created query plans for each query using both the selectivity- and coherence-based reordering strategies. Each query plan is then run entirely live, entirely against the store, and also run for every possible split position in both orders where part goes live and part goes to the store. This allows to analyse the effect of different strategies by simply computing the split position without the need to rerun the query.

MEASURES We obtained the *speed-up* and *recall* compared to a pure live execution for each query and the different hybrid query plans and split position, but also for the execution of the query against the cache, Specifically, to calculate speed up, the total time taken by the live approach to run all queries is divided by the total time taken for each individual approach; e.g., a speed up of 6 indicates that the approach in question was 6× faster than live querying. Conversely, recall is measured by taking live querying as the gold standard.

NOTATIONS An overview about the notation used for the different ordering and splitting approaches for the discussion of the results is presented in Table 6.3. We intuitively denote the different orderings with a `sel-*` prefix for the selectivity based and a `coh-*` prefix for coherence based. The best split approaches are represented by `*-best`; these splits cannot be determined before query execution, but rather represent the ideal case. Splitting at the most incoherent pattern is indicated by `*-incoh`. Using a coherence threshold of 0.5 to perform the split is indicated by `*-thr`. A random split (i.e., a guess) is indicated by `*-rnd`. Fixed split positions are indicated by `*-1` and `*-2` for $n = 1, 2$. Note that the threshold strategies

store	*stable*$_{(\checkmark)}$	*local error*$_{(X)}$	*no live*$_{(X)}$	*total*
OpenLink	98	18	84	200
Sindice	91	25	84	200

Table 6.4: Statistics about evaluation queries per store

	avg. deviation	
approach	recall	time
LTBQE	3%	2.7%
OpenLink	0–5%	2–36%
Sindice	0–2%	1–17%

Table 6.5: Average time and recall deviation for all queries across four runs/

`*-incoh`/`*-thr` can go fully live or fully local depending on the coherence values found for the query, whereas `*-best`, `*-rnd`, `*-1` and `*-2` must split the query.

6.4.2 Results

Given that we run queries over remote data and public accessible SPARQL stores, we may encounter unstable behaviour. Thus, we reran all experiments four times over a period of eight days. Despite our precautions, as summarised in Table 6.4, we could not use all of the original 200 queries for our evaluation. First, although our query generation algorithm is designed to only derive queries that LTBQE can answer, 84 of the queries would return no live results, possibly due to changes in remote data since the BTC'11 dataset was crawled, or due to prolonged down-times in remote sources. This provides even more evidence of the dynamicity of the Web data. Second, 18 queries for OpenLink and 25 for Sindice returned an error (e.g., memory exceptions, timeouts, `50x` response codes) in all four runs for all configurations involving the store. We exclude these queries from subsequent analysis, though it should be noted that we end up filtering over half of the original queries. Our benchmark results thus refer to 98 stable queries for the OpenLink endpoint and 91 stable queries for the Sindice endpoint.

REPEATABILITY In terms of the repeatability of results, for each configuration, we measured deviations for recall of results and query time across the four runs vs. the best approach (highest recall, lowest time). We then averaged the deviations across all queries. Table 6.5 shows the obtained values. We measured an average recall deviation of 3% and a time deviation of 2.7% for the LTBQE approach. For the various configurations, the recall deviation varied between 0–5% for OpenLink and between 0–2% for the Sindice endpoint. Although the recall of the endpoints was very stable, we observed average time deviations of up to 36% for the Openlink and 17% for the Sindice endpoint, indicating variable query response times. Here acknowledging that public endpoints and remote data sources can be unstable, we henceforth wish to factor out this instability to derive comparable results across different hybrid strategies: our focus herein is to evaluate and compare different hybrid query plans, not the performance of public SPARQL stores. Thus, to avoid outliers, for each query and each configuration, we only select the best run in terms

improvement	OpenLink		Sindice	
	`sel`	`coh`	`sel`	`coh`
Better Than Local Recall:	43%	53%	87%	91%
Better Than or Equal Local Recall:	97%	100%	99%	97%
Better Than Live Time:	92%	45%	16%	3%
Better Than Local Recall & Live Time	39%	13%	8%	1%
Better Than Or Equal Local Recall & Live Time:	92%	45%	16%	3%

Table 6.6: For both stores, the percentage of queries that can potentially be improved for each order assuming the best split position is picked.

of recall, and in case two runs have the same recall, we select the one with the lower query time.

6.4.2.1 *Proof of Concept*

To initially prove concept, we first want to show that, in practice, hybrid query execution can *potentially* improve the recall of fresh results over the store while reducing the time taken for the live approach.

Table 6.6 presents such an analysis for both stores, where we see the potential percentage of queries that can be improved using our hybrid approach for both orders. For these results we assume that the best possible split position is picked (i.e., given the results, we select the split position that gave the highest recall and if tied, the lowest time; we evaluate split-selection strategies later). Recall is measured relative to the entirely live results, which we know to be fresh. For OpenLink, we see that the recall of the store can only be improved for roughly half of the queries; however, the recall of the store is already 1 in many cases and cannot be improved, only equalled. Note that ties in time are much more rare. In terms of improving the time for live results, the `sel` ordering seems much more beneficial for OpenLink than `coh`, likely due to fewer intermediate results being generated in the former ordering: `sel` improves or equals the local recall while improving the live time in 92% of the queries. For Sindice, we found that the store often returned no query results: 84% of the queries ran entirely live as a fallback. Thus, the recall of many queries can be improved outright, but few queries are faster than the live approach. Since only 16% of the queries for Sindice are run in a truly hybrid fashion, we henceforth focus on OpenLink. All hybrid query results for Sindice were very close to the live approach. Table 6.6 shows that, in an ideal case, the hybrid approach can indeed improve result freshness while reducing the time required to process queries. Further, the chosen strategy seems to have a clear impact on the achieved freshness and query time.

6.4.2.2 *Different splits and ordering combinations*

Of course, Table 6.6 does not tell the whole story for OpenLink, but rather gives a quantitative validation for the improvements theoretically possible through hybrid querying. Crucially, we are still left to determine a strategy to find the optimal split for each ordering, and we have yet to see the *degree* to which local recall is improved and live querying is sped up. To compare different splits and ordering combinations, we first filter out queries that, across the four runs, did not provide results for all possible split positions and orderings for one or more of the setups. We also removed queries with only two patterns, for which the choice of split is trivial. This results in a final set of 43 queries.

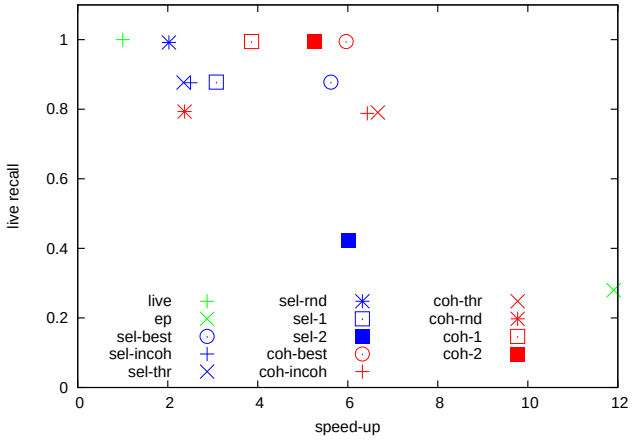


Figure 6.5: Recall vs. speed-up trade-off for all different hybrid plans with `thr` = 0.5

For each ordering and for a variety of different split strategies, Figure 6.5 plots the aggregate speed up and recall ratio versus live querying. Figure 6.6 shows the same analysis, but for varying coherence threshold values. Based on our approach to compute recall and speed-up, the live querying approach is placed at point $(1, 1)$ in the figure. In fact, both plots offer an empirical version of the trade-off introduced in Figure 6.1, where our hybrid strategies sit between live querying and the store. We can see, that non of the hybrid approaches was in average slower than the live approach. For both graphs, we see that the store is the fastest, and about $12\times$ faster than the live approach; however, the recall of the store is poor with around a third of the recall of the live approach. Looking at the performance of the hybrid approaches in Figure 6.5, we see that the selectivity based ordering approaches (colored blue) have in general a higher recall but lower speed-up than coherence based ordering approaches (colored red). From the former, 5 out of 6 approaches achieve an average recall of above 80%, whereas 3 out of the 6 coherence based approaches are below this 80% recall mark. Regarding the speed-up, we see exactly the opposite, with 5 out of 6 coherence based approaches showing a speed-up of more than $4\times$, whereas 3 out of the 6 `sel-*` approaches are below this speed-up mark. However, the most promising hybrid query planning is guided by the coherence measures if we look at the approaches which have the best trade-off between recall and speed-up. The best trade-off of a full recall and a $7\times$ speed-up is achieved by `coh-best`. Also the best speed-up and also recall is achieved by planning approaches guided by the coherence measures. Interestingly, a fixed split position as in `coh-2` in Figure 6.5, which is $\sim 5\times$ faster than live, but maintains an almost perfect recall, can approach the ideal of `coh-best` quite closely. Looking closer at Figure 6.6, we can see that perhaps the best hybrid approach is `coh-thr=0.75` in, which maintains an almost perfect recall but offers a speed up of more than $6\times$ live querying, slightly beating the ideal of `coh-best` (which must definitely split the query).

6.4.2.3 *Recall and speed-up per query*

While such an aggregated view presents very interesting insights into the overall performance of the different strategies, we cannot identify the distribution over the queries in terms of achieved freshness and time. This is supported by Figure 6.7 which shows the recall per query for the different approaches and by Figure 6.8the query time for each query respectively. For both figures, we plot the number of queries on the x-axis that achieve a certain recall or time ratio shown on the y-

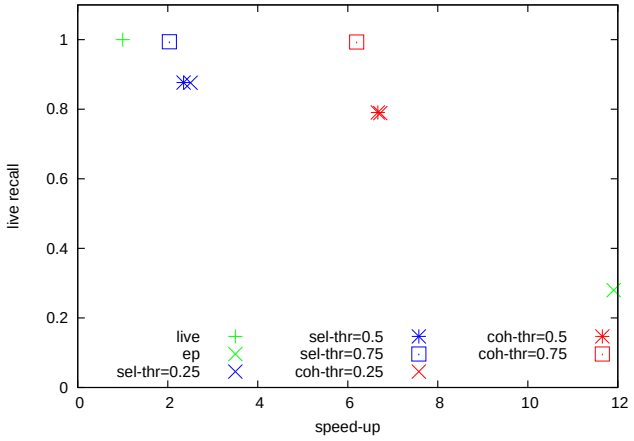


Figure 6.6: Recall vs. speed-up trade-off for different thresholds

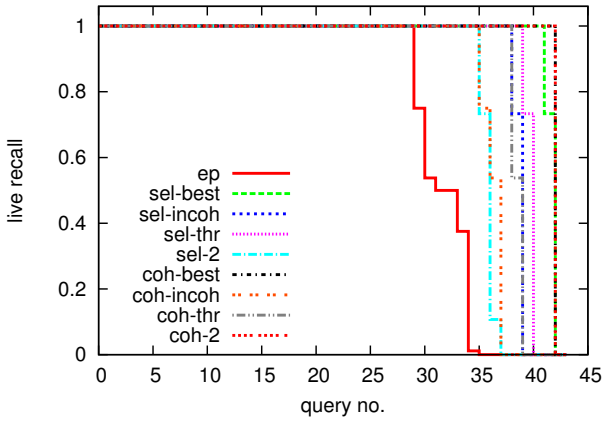


Figure 6.7: Queries ordered by *recall* for our different order and split strategies

axis. Further, all 43 queries are sorted for each approach separately. This provides us with a global view on each performance, but does not yet support a per-query comparison of the strategies.

RECALL Figure 6.7 shows that querying only the centralised store results in the fewest number of queries with a recall of 1, and also results in the most queries with a recall between 0 and 1. On the contrary, `coh-best` and `coh-2` are tied for keeping 100% recall across 42 queries and provide 0% only for the last query. This aligns with the results shown in Figure 6.5. Interestingly, most queries run with any hybrid strategy result in a recall of either 1 or 0.

TIME RATIO Figure 6.8 shows the time ratio of the different approaches compared to the live approach. As expected, the time required for querying the store is far below all other approaches in most cases. However, it is in fact slower for 2 queries. We found that this was due to some anomalous queries that consistently returned slow results.[6] We see that the approaches `sel-incoh` and also `sel-thres` did not improve the time of the live approach for around 13 queries, which means

6 One such example at the time of writing was `http://bit.ly/IPRec9`.

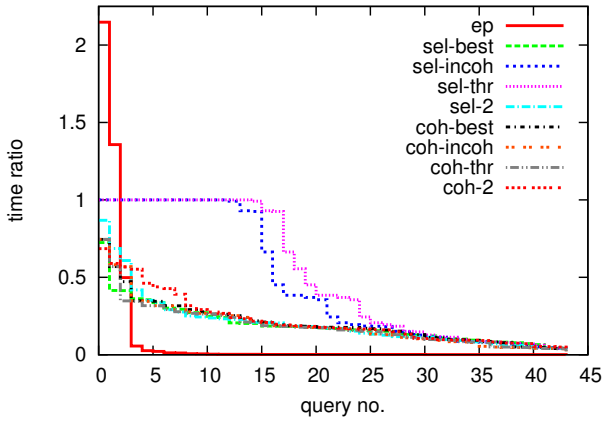


Figure 6.8: Queries ordered by *time* for different order and split strategies

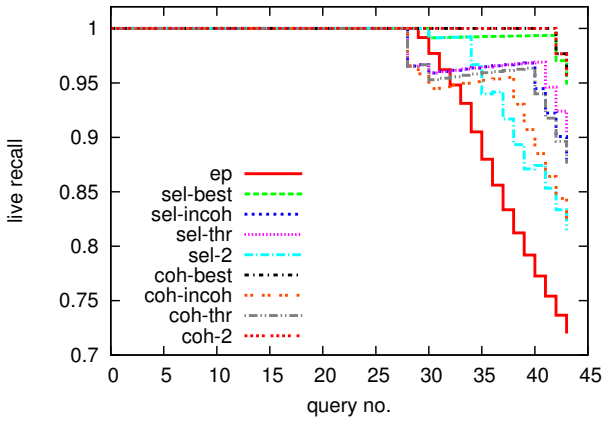


Figure 6.9: Evolving average for *query recall* with different order and split strategies

that the determined split position indicated that the queries should be run completely live. All other strategies show very similar overall performance by means of query time. They decrease the time of the live approach for all queries.

6.4.2.4 *Per-query comparison of strategies*

Eventually, we are interested in how recall and query times compare for different approaches for the same queries. Thus, we ordered the queries for each approach identically, using the results of the store approach as basis. This means that in these figures each point on the x-axis presents the *same* query for each approach. In this case, plotting the absolute values would not allow any meaningful insights due to the ups and downs that each plot would show. Instead, we plot an "evolving average", whereby the result for query n indicates the average value for all queries up to and including n. This allows to compare the degree of increase or decrease in recall and time at each point, i.e., for each query. Figure 6.9 shows the recall values for each approach and query (note that the y-axis is zoomed in on $[0.7, 1.1]$). Figure 6.10 shows the "evolving average" of the time ratio across the queries.

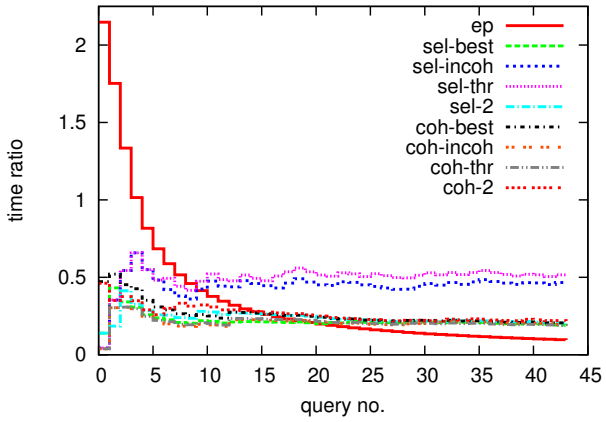


Figure 6.10: Evolving average for *query time* with different order and split strategies

RECALL Interestingly, we see that the store can sometimes return better recall than the hybrid approaches, as happens for query 27. A manual inspection of the execution of the query revealed that the interim bindings returned by OpenLink cannot be dereferenced. In that case, the live approach fails to collect query relevant information and thus misses results. However, the hybrid approaches improve the recall for the subsequent queries.

Furthermore, it is interesting to observe that the hybrid approaches seem to be "grouped". We can see that `coh-best`, `coh-2` and `sel-best` show very similar performance over all queries. Whereas, `coh-incoh` first "follows" `sel-thr` and others, but performs similar to `sel-2` for later queries.

TIME RATIO The evolving average of the query times in Figure 6.10 basically confirm the results from Figure 6.8. The store query time evolves to be the fastest after around 20 queries. Moreover, we can see the same two groups as in Figure 6.8, consisting of `sel-incoh` and `sel-thr` in the one and the remaining hybrid approaches in the other group.

6.5 CONCLUSION

In summary, we found that sending more patterns live with fewer bindings (as with the selectivity-based ordering) is in parts faster than sending less patterns live with more bindings (coherence-based ordering). The fact that more of the query is executed live compensates for that fact that the coherence estimates are not considered in the ordering; in some cases it can even help to overcome estimate errors. The lower query times suggested by the coherence-based orderings are often compensated by the low selectivities of operators in the lower levels of query plans. Still, as a guideline, if the objective is to maximise recall, one should choose coherence-based ordering, which is still faster then the live approach. If one is willing to sacrifice some recall for even faster results, then selectivity-based ordering is a good choice. However, the performance of the selectivity-based approaches seems to heavily depend on the actually chosen split strategy. Generally, the question of how to pick the split position cannot be ultimately answered without taking the actual query and other characteristics into account. While the actual value of the coherence threshold did not have an impact as high as expected, we could show that the coherence estimates themselves are of great benefit.

We see this combination of live and materialised querying approaches as an important development for Linked Data. Herein, we may only have scratched the surface of what is possible, essentially validating the core idea of hybrid query execution and demonstrating the inclusion of coherence dynamicity estimates into query planning. We hope to see further works expanding upon this central theme in the future. We elaborate further on future direction later in Section 7.3.

7 CONCLUSION

"Keep up the spirit for the last mile!"

— Axel Polleres, *2012*

Semantic Web technologies and the Linked Data guidelines bear the potential to transform the Web into a network of knowledge which can be efficiently processed by machines. Some crucial requirements in this movement are efficient techniques to search and query relevant information, which becomes (even more) challenging when query relevant data is dynamic and up-to-date/fresh query results are expected.

The presented thesis was motivated by the observation that existing query approaches for Linked Data facilitate either fast query times (by using optimised centralised stores) or up-to-date results (by processing a query directly online over the Web). However, there existed no solution to efficiently query dynamic Linked Data and guarantee up-to-date results at the same time.

7.1 CONTRIBUTIONS

(*contribution to* H1)

We have confirmed that a significant share of Linked Data is dynamic – around 40% of the monitored sources changed at least once during the four month experiment we performed and 18% changed at least every week. We then observed that centralised search engines (based on data replication) cannot always guarantee up-to-date query results due to the dynamicity of Linked Data.

We systematically investigated alternative query approaches which guarantee up-to-date results in the remainder of this thesis.

(*contribution to* H2 and H3)

We have studied pure linked-traversal based query execution approaches, which guarantee up-to-date results but pose practical restrictions on the type of executable queries and suffer from potentially low result recall. Our experiments have shown that the selection of sources can be successfully reduced, without influencing the query results, by ignoring predicate URIs, object URIs for type-triples, and URIs bound to non-join positions. Furthermore, we have extended the standard link traversal based query execution approach to exploit the semantics of a subset of the RDFS rules and of `owl:sameAs` equality statements. Our comprehensive evaluation showed that the `owl:sameAs` extension can increase the number of results by 50% for 20% of our query classes, but also comes at significant costs and introduces unstable behaviour when executed live over domains such as DBpedia. Similarly, we observed that RDFS reasoning increases results more frequently (e.g., in around 80% of the query classes) than `owl:sameAs` extensions (e.g., in lower percentiles of the QWalk experiments), but exhibits more moderate increases than the latter extensions (e.g., in the 100^{th} percentiles of QWalk experiments).

Overall, we can conclude that linked-traversal based query execution works well for simple queries with a dereferenceable subject, but, in uncontrolled environments, struggles for more complex queries that involve accessing many remote sources at runtime.

(*contribution to* H4)

We have compared in-depth various source selection approaches to loosen the query type restriction posed by pure link traversal based approaches. As a result, we developed a hash-based index structure to summarise the graph-structured data (RDF), provided algorithms for processing conjunctive SPARQL queries and included a source ranking to further control the query time. Experimental results showed that our lightweight hash-based index structure returns potentially faster and more complete results compared to other studied approaches.

Both approaches (link traversal and data summaries) are complementary and offer the potential to get fresher answers than centralised approaches and to discover new sources when dynamic information is involved. However, retrieving remote content from diverse sources at query-time naturally implies a slower response times compared to optimised local indexes with data replicated from (parts of) the Web.

(*contribution to* H5)

To obtain the best of both worlds, we proposed and developed a hybrid query framework that combines centralised and distributed query approaches. Our engine features a novel query planner that decides which parts of the query to answer directly over the Web and which query parts to run locally. The query planner splits a query into a local and live part based on knowledge how coherent/up-to-date a specific centralised index is with respect to the given query patterns against remote Web data. The dynamicity knowledge also involves both the dynamicity of remote data and the coverage of the centralised index. Our results show that hybrid query execution can indeed improve freshness vs. fully cached results while reducing the time taken vs. fully live execution when compared with linked-traversal based query execution, the hybrid approach in average maintains an almost perfect recall while offering a significant speed up.

7.2 LESSONS LEARNT

During the initial designs for our experiments to study the dynamicity of Linked Data and the coherence of centralised public SPARQL stores, we realised that it is far from trivial to sample a representative collection of Linked Data sources due to the current uptake of Linked Data. The number of data providers continuously increases and new data becomes available every months. This makes it hard to decide which sources can be seen as generally representative. Furthermore, to perform interesting experiments and derive meaningful conclusion it is necessary to have a complete history of snapshots for the specific monitoring intervals. However, this requires the availability of several resources, such as storage space, backup solutions and stable bandwidth. Ideally, the effort is community driven to collect a broad range of use cases and coordinated by several partners. We currently try to establish such a community and started a dynamic Linked Data observatory.

We recognised during the evaluation of our approaches that the efficiency of a query approach for the Web does not only depend on the chosen algorithms and index structures, but it is also influenced by external factors such as the available bandwidth, politeness guidelines and data provider resources. We also emphasise that runtimes in uncontrolled environments are often influences by external factors like “politeness policies”, for instance when queries often touch upon documents from the same domain. There are further crucial issues with the hosting reliability of data providers.

Another issue we had to face was the lack of proper benchmarks for decentralised Linked Data query approaches. Many of the available benchmarks are de-

signed to evaluate centralised approaches in a controlled environment. We overcome this challenge by designing our own query generator which allowed us to create large numbers of queries in various shapes. These random generated queries are well suited to systematically evaluate our approaches but do not necessarily reflect the query types and use the datasets of real-world applications. Ideally, one would like to have a collection of queries with different complexity, from various domains and which are actually used in applications.

Ultimately, we believe that our hybrid query approach makes a significant step towards a new generation of Linked Data query engines by combining and exploiting the strength of different query approaches. Given the potential scope and dynamicity of Linked Data, we are believe that the next generation of query engines will need to deploy a range of techniques to efficiently offer fresh results with broad coverage. Our algorithms are (to provide) the first steps in this direction.

7.3 FUTURE DIRECTIONS

In this thesis, we performed studies to confirm the dynamicity of Linked Data, investigated query approaches which can operate over the dynamic data and developed a new type of hybrid query engine. We now highlight what we believe to be important future directions one can take from here.

7.3.1 Dynamic Linked Data Observatory

We know from a plethora of published studies that the traditional Web is highly dynamic and that the various findings about the dynamic processes can have an essential impact in developing optimisations for tasks such as web crawling and caching [Cho and Garcia-Molina, 2003b], maintaining link integrity [Popitsch and Haslhofer, 2011], servicing of continuous queries [Pandey et al., 2003] or for replication and synchronisation task [Tummarello et al., 2007]. In contrast to the well studied Web, the change and creation processes on the Web of Data are still an almost entirely unexplored field and most of the research is based on little knowledge or on the assumptions that similar dynamic characteristics can be found as on the traditional Web.

We contribute to this research area by providing access to the datasets collected by our Dynamic Linked Data Observatory, which we featured in Section 3.2 and which is described in detail by Käfer et al. [2012]. The collection of weekly snapshots allows the community to study not only high level dynamics of sources (as we did in Section 3.2) but also to study dynamics between sources, e.g., by applying spatio-temporal correlation analysis on linking patterns to uncover root causes of change to the link structure of the Web graph as we identified earlier [Umbrich et al., 2010b].

We are convinced that a comprehensive understanding of the dynamics of the Web of Data is a necessary requirement to efficiently develop algorithms and frameworks to deal with the dynamic Web of Data. This will be even more important, if we consider that in the future even more dynamic data sources will be part of the Web of Data (e.g., sources which publish information about temperature, location, seismic activities or user content from moving mobile devices).

7.3.2 On Hybrid Querying

Ultimately, we are convinced that a hybrid query model, such as the one proposed herein, is the a very promising approach to efficiently query the dynamic Web of

Data. While we believe in the proposed approach that combines live Web queries with the access to centralised repositories, we understand and highlighted the wide range of challenges it bears. However, the first steps that we take in order to overcome these challenges do serve to validate the feasibility of the proposed architecture.

We have looked at a wide variety of configurations which hint at the potential complexity of hybrid query planning. More complex cost models – including, e.g., the potential for multiple splits as discussed in Section 6.3 – may reveal novel optimisations that we have not yet considered herein, further pushing the boundaries of fresh vs. fast results. Furthermore, we can only estimate the accuracy of endpoint results using coherence estimates; other mechanisms that cross-check the sources of data (i.e., the named graphs) from which the endpoint computes answers against their current versions could yield more accurate statistics. A mix between push based (e.g. ping services) and pull based (e.g. continuos crawling) can also result in efficient strategies to learn, discover and update knowledge about changes. Further experiences from research about (Web) caching [Douglis et al., 1997] and replication, in conjunction with the results from mining Web data [Umbrich et al., 2010b], will have major impact on the chosen combination of these methods.

Also, in our hybrid framework, we assume that the live and index query components are strongly decoupled, which allows to use any third party query provider such as the studied public SPARQL endpoints (cf. Chapter 6). One can even think of a tight integration between the live query processor and the used repository and develop a query engine which has direct access to the optimised local index combined with an integrated LTBQE engine. In addition, the local index may serve as a source selection index to enhance the live results given by a zero-knowledge approach such as LTBQE (similar to [Ladwig and Tran, 2011]).

7.3.3 Towards a Query Language for the Web

We see from the results in Chapter 4 that LTBQE cannot be considered a complete solution for running complex SPARQL queries over Linked Data due to various fundamental (e.g., no support for `OPTIONAL`, *etc.*) and practical issues like the:

- reliance on dereferenceability of URIs,
- assumptions that query-patterns connect relevant sources through dereferenceable URIs,
- slow access to remote sources, and
- varying stability of remote hosts.

SPARQL is simply too complex a query language to be supported in its entirety and in a practical fashion by LTBQE. As such, one may consider different languages for navigational queries. In general, a query language that would allow for declaratively specifying navigational aspects of query execution – e.g., stick to the `data.semanticweb.org` domain, follow `foaf:knows` links, do not follow `foaf:homepage` links, etc. – would be interesting, and would allow users to better guide the query-engine than using a simple SPARQL query.

Along these lines, various authors have also questioned whether SPARQL is the right language to query the Web of Data: again, SPARQL is defined for closed datasets and was originally proposed with materialised settings in mind. Relatedly, there have been a number of proposals to extend SPARQL with regular expressions that capture navigational patterns, including work by Alkhateeb et al. [2009] and work on the nSPARQL language [Pérez et al., 2010]. SPARQL 1.1 includes a similar

notion called *property paths* [Harris and Seaborne, 2012] which we have not yet investigated in the context of LTBQE. Recently, Fionda et al. [2012] proposed NautiLOD, a novel declarative language for navigating paths in the Web of Data guided by regular expressions over RDF predicates, using SPARQL `ASK` queries to *test* some conditions over the data encountered (i. e., to find data matching a query), and allowing to trigger some actions whenever some condition is met. Such work goes beyond pure SPARQL querying, but perhaps touches upon some of the broader potential of consuming the Web of Data in a declarative manner.

BIBLIOGRAPHY

Aberer, K., Choi, K.-S., Noy, N. F., Allemang, D., Lee, K.-I., Nixon, L. J. B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., and Cudré-Mauroux, P., editors (2007). *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, Busan, Korea. Springer.

Aberer, K., Cudré-Mauroux, P., Hauswirth, M., and Pelt, T. V. (2004). Gridvine: Building internet-scale semantic overlay networks. In [McIlraith et al., 2004], pages 107–121.

Adjiman, P., Goasdoué, F., and Rousset, M.-C. (2006). SomeRDFS in the Semantic Web. *Journal on Data Semantics*, 8. Available from: `http://hal.inria.fr/inria-00091169`.

Alexander, K. and Hausenblas, M. (2009). Describing linked datasets - on the design and usage of voiD, the 'vocabulary of interlinked datasets. In *Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09*.

Alkhateeb, F., Baget, J.-F., and Euzenat, J. (2009). Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73.

Andersen, P. B. (1998). WWW as a self-organizing system. *Cybernetics & Human Knowing*, pages 5 – 41.

Antoniou, G., Antoniou, G., Antoniou, G., Harmelen, F. V., and Harmelen, F. V. (2003). Web ontology language: Owl. In *Handbook on Ontologies in Information Systems*, pages 67–92. Springer.

Antoniou, G., Grobelnik, M., Simperl, E. P. B., Parsia, B., Plexousakis, D., Leenheer, P. D., and Pan, J. Z., editors (2011). *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*, volume 6643 of *Lecture Notes in Computer Science*, Crete, Greece. Springer.

Aranda, C. B., Arenas, M., and Corcho, Ó. (2011). Semantics and optimization of the SPARQL 1.1 federation extension. In [Antoniou et al., 2011], pages 1–15.

Arasu, A. and Garcia-Molina, H. (2003). Extracting structured data from web pages. In Halevy, A. Y., Ives, Z. G., and Doan, A., editors, *SIGMOD Conference*, pages 337–348. ACM.

Arias, M., Fernández, J. D., Martínez-Prieto, M. A., and de la Fuente, P. (2011). An empirical study of real-world SPARQL queries. *CoRR*, abs/1103.5043.

Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N. F., and Blomqvist, E., editors (2011). *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, Bonn, Germany. Springer.

Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. G. (2007). Dbpedia: A nucleus for a web of open data. In [Aberer et al., 2007], pages 722–735.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In Popa, L., Abiteboul, S., and Kolaitis, P. G., editors, *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM.

Baeza-Yates, R. A., Ciaramita, M., Mika, P., and Zaragoza, H. (2008). Towards semantic search. In *NLDB*, volume 5039 of *Lecture Notes in Computer Science*, pages 4–11. Springer.

Bechhofer, S., Hauswirth, M., Hoffmann, J., and Koubarakis, M., editors (2008). *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, volume 5021 of *Lecture Notes in Computer Science*, Tenerife, Canary Islands, Spai. Springer.

Beckett, D. and Berners-Lee, T. Turtle - terse RDF triple language, W3C team submission [online]. (2008). Available from: `http://www.w3.org/TeamSubmission/turtle/`. See:.

Berners-Lee, T. Semantic Web road map [online]. (1998). Available from: `http://www.w3.org/DesignIssues/Semantic.html`.

Berners-Lee, T. Linked Data - design issues [online]. (2006). Available from: `http://www.w3.org/DesignIssues/LinkedData.html`. http://www.w3.org/DesignIssues/LinkedData.html.

Berners-Lee, T. and Cailliau, R. WorldWideWeb: Proposal for a HyperText Project [online]. (1990) [cited 07-01-2011]. Available from: `http://www.w3.org/Proposal.html`.

Berners-Lee, T., Fielding, R. T., and Masinter, L. Uniform Resource Identifier (URI): Generic Syntax [online]. (2005). Available from: `http://tools.ietf.org/html/rfc3986`.

Birbeck, M. and McCarron, S. Curie syntax 1.0 — a syntax for expressing compact uris [online]. (2008). Available from: `http://www.w3.org/TR/2008/WD-curie-20080506`.

Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., and Velkov, R. (2011a). Owlim: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42.

Bishop, B., Kiryakov, A., Ognyanov, D., Peikov, I., Tashev, Z., and Velkov, R. (2011b). FactForge: A fast track to the Web of Data. *Semantic Web*, 2(2):157–166.

Bizer, C., Cyganiak, R., and Heath, T. How to publish Linked Data on the Web [online]. (2007). Available from: `http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/`.

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked Data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22.

Bizer, C., Jentzsch, A., and Cyganiak, R. State of the lod cloud [online]. (2011) [cited 05-04-2011]. Available from: `http://www4.wiwiss.fu-berlin.de/lodcloud/state/`.

Bizer, C. and Maynard, D. (2011). The Semantic Web challenge, 2010. *J. Web Sem.*, 9(3):315.

Bizer, C. and Schultz, A. (2009). The Berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24.

Bonatti, P. A., Hogan, A., Polleres, A., and Sauro, L. (2011). Robust and scalable Linked Data reasoning incorporating provenance and trust annotations. *J. Web Sem.*, 9(2):165–201.

Bouquet, P., Ghidini, C., and Serafini, L. (2010). A formal model of queries on interlinked RDF graphs. In *AAAI Spring Symposium: Linked Data Meets Artificial Intelligence*. AAAI.

Brewington, B. E. and Cybenko, G. (2000a). How dynamic is the Web? *Computer Networks*, 33(1-6):257–276.

Brewington, B. E. and Cybenko, G. (2000b). Keeping up with the changing web. *IEEE Computer*, 33(5):52–58.

Brickley, D. and Guha, R. V. (2004). RDF vocabulary description language 1.0: RDF schema. Technical report, W3C. Available from: `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. In *Computer Networks and ISDN Systems*, pages 107–117.

Bruno, N., Chaudhuri, S., and Gravano, L. (2001). Stholes: A multidimensional workload-aware histogram. In *SIGMOD Conference*, pages 211–222.

Buitelaar, P. and Cimiano, P., editors (2008). *Ontology Learning and Population: Bridging the Gap between Text and Knowledge*, volume 167 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam.

Cai, M. and Frank, M. R. (2004). RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In [Feldman et al., 2004], pages 650–657.

Chakrabarti, K., Garofalakis, M. N., Rastogi, R., and Shim, K. (2001). Approximate query processing using wavelets. *VLDB J.*, 10(2-3):199–223.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2).

Chen, W., Naughton, J. F., and Bernstein, P. A., editors (2000). *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM.

Cheng, G. and Qu, Y. (2008). Term dependence on the Semantic Web. In [Sheth et al., 2008], pages 665–680.

Cho, J. and Garcia-Molina, H. (2000a). The evolution of the web and implications for an incremental crawler. In Abbadi, A. E., Brodie, M. L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., and Whang, K.-Y., editors, *VLDB*, pages 200–209. Morgan Kaufmann.

Cho, J. and Garcia-Molina, H. (2000b). Synchronizing a database to improve freshness. In [Chen et al., 2000], pages 117–128.

Cho, J. and Garcia-Molina, H. (2003a). Effective page refresh policies for Web crawlers. *ACM Trans. Database Syst.*, 28(4):390–426.

Cho, J. and Garcia-Molina, H. (2003b). Estimating frequency of change. *ACM Trans. Internet Techn.*, 3(3):256–290.

Crespo, A. and Garcia-Molina, H. (2002). Routing indices for peer-to-peer systems. In *ICDCS*, pages 23–.

Cruz, I. F., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., and Aroyo, L., editors (2006). *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, Athens, GA, USA. Springer.

Cudré-Mauroux, P., Agarwal, S., and Aberer, K. (2007). Gridvine: An infrastructure for peer information management. *IEEE Internet Computing*, 11(5):36–44.

Cyganiak, R., Hogan, A., and Harth, A. N-Quads: Extending N-Triples with context [online]. (2008). Available from: `http://sw.deri.org/2008/07/n-quads/`.

de Kunder, M. The size of the world wide web (the internet) [online]. (2012) [cited 2012-06-08]. Available from: `http://www.worldwidewebsize.com/`.

Dean, M. and Schreiber, G. (2004). OWL web ontology language reference. W3C recommendation, W3C.

Decker, S., Mitra, P., and Melnik, S. (2000). Framework for the Semantic Web: An RDF tutorial. *IEEE Internet Computing*, 4(6):68–73.

Delbru, R. (2009). Siren: entity retrieval system for the web of data. In *Proceedings of the Third BCS-IRSG conference on Future Directions in Information Access*, FDIA'09, pages 29–35, Swinton, UK, UK. British Computer Society. Available from: `http://dl.acm.org/citation.cfm?id=2227296.2227302`.

Delbru, R., Campinas, S., and Tummarello, G. (2012). Searching web data: An entity retrieval and high-performance indexing model. *J. Web Sem.*, 10:33–58.

Delbru, R., Toupikov, N., Catasta, M., and Tummarello, G. (2010). A node indexing scheme for web entity retrieval. In Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., and Tudorache, T., editors, *ESWC (2)*, volume 6089 of *Lecture Notes in Computer Science*, pages 240–256, Crete, Greece. Springer.

Delbru, R., Tummarello, G., and Polleres, A. (2011). Context-dependent OWL reasoning in Sindice - experiences and lessons learnt. In Rudolph, S. and Gutierrez, C., editors, *RR*, volume 6902 of *Lecture Notes in Computer Science*, pages 46–60. Springer.

Ding, L. and Finin, T. (2006). Characterizing the Semantic Web on the Web. In [Cruz et al., 2006], pages 242–257.

Ding, L., Shinavier, J., Shangguan, Z., and McGuinness, D. L. (2010). SameAs networks and beyond: Analyzing deployment status and implications of owl:sameas in Linked Data. In [Patel-Schneider et al., 2010], pages 145–160.

Domenig, R. and Dittrich, K. R. (1999). An overview and classification of mediated query systems. *SIGMOD Record*, 28(3):63–72.

Douglis, F., Feldmann, A., Krishnamurthy, B., and Mogul, J. C. (1997). Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internet Technologies and Systems*.

Erling, O. and Mikhailov, I. (2009). Virtuoso: RDF support in a native RDBMS. In de Virgilio, R., Giunchiglia, F., and Tanca, L., editors, *Semantic Web Information Management*, pages 501–519. Springer.

Feldman, S. I., Uretsky, M., Najork, M., and Wills, C. E., editors (2004). *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*. ACM.

Fetterly, D., Manasse, M., Najork, M., and Wiener, J. L. (2004). A large-scale study of the evolution of web pages. *Softw., Pract. Exper.*, 34(2):213–237.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext transfer protocol – http/1.1 [online]. (1999). Available from: `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

Fionda, V., Gutierrez, C., and Pirrò, G. (2012). Semantic navigation on the web of data: specification of routes, web fragments and actions. In Mille, A., Gandon, F. L., Misselis, J., Rabinovich, M., and Staab, S., editors, *WWW*, pages 281–290. ACM.

Fuchs, C. (2005). The Internet as a self-organizing socio-technological system. *Cybernetics & Human Knowing*, 12(3):37–81. Available from: `http://www.ingentaconnect.com/content/imp/chk/2005/00000012/00000003/art00004`.

Gibbons, P. B., Matias, Y., and Poosala, V. (2002). Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.*, 27(3):261–298.

Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., and Strauss, M. (2001). Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In Apers, P. M. G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., and Snodgrass, R. T., editors, *VLDB*, pages 79–88. Morgan Kaufmann.

Glimm, B., Hogan, A., and Krötzsch, M. (2012). OWL: Yet to arrive on the Web of Data? In *Linked Data on the Web Workshop (at WWW2012)*.

Goldman, R. and Widom, J. (1997). Dataguides: Enabling query formulation and optimization in semistructured databases. In [Jarke et al., 1997], pages 436–445.

Görlitz, O. and Staab, S. (2011). SPLENDID: SPARQL endpoint federation exploiting voiD descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data*, Bonn, Germany.

Gunopulos, D., Kollios, G., Tsotras, V. J., and Domeniconi, C. (2000). Approximating multi-dimensional aggregate range queries over real attributes. In [Chen et al., 2000], pages 463–474.

Guo, Y., Pan, Z., and Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In Yormark, B., editor, *SIGMOD Conference*, pages 47–57. ACM Press.

Halpin, H., Hayes, P. J., McCusker, J. P., McGuinness, D. L., and Thompson, H. S. (2010). When owl:sameAs isn't the same: An analysis of identity in Linked Data. In [Patel-Schneider et al., 2010], pages 305–320.

Harris, S. and Seaborne, A. SPARQL 1.1 query language [online]. (2012). Available from: `http://www.w3.org/TR/sparql11-query/`.

Harth, A. and Decker, S. (2005). Optimized index structures for querying RDF from the Web. In *LA-WEB*, pages 71–80. IEEE Computer Society.

Harth, A., Hogan, A., Delbru, R., Umbrich, J., O'Riain, S., and Decker, S. (2007a). SWSE: Answers before links! In Golbeck, J. and Mika, P., editors, *Semantic Web Challenge*, volume 295 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.-U., and Umbrich, J. (2010). Data summaries for on-demand queries over Linked Data. In Rappa, M., Jones, P., Freire, J., and Chakrabarti, S., editors, *WWW*. ACM.

Harth, A. and Maynard, D. The billion triple challenge [online]. (2012). Available from: `http://challenge.semanticweb.org/`.

Harth, A., Umbrich, J., and Decker, S. (2006). Multicrawler: A pipelined architecture for crawling and indexing Semantic Web data. In [Cruz et al., 2006].

Harth, A., Umbrich, J., Hogan, A., and Decker, S. (2007b). YARS2: A federated repository for querying graph structured data from the Web. In [Aberer et al., 2007], pages 211–224.

Hartig, O. (2011a). How caching improves efficiency and result completeness for querying Linked Data. In *Proceedings of the 4th Linked Data on the Web (LDOW) Workshop at the World Wide Web Conference (WWW)*, Hyderabad, India.

Hartig, O. (2011b). Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In [Antoniou et al., 2011], pages 154–169.

Hartig, O. (2012). SPARQL for a Web of Linked Data: Semantics and computability. In [Simperl et al., 2012], pages 8–23.

Hartig, O., Bizer, C., and Freytag, J. C. (2009). Executing SPARQL queries over the Web of Linked Data. In Bernstein, A., Karger, D. R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., and Thirunarayan, K., editors, *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 293–309, Chantilly, VA, USA. Springer.

Hartig, O. and Freytag, J. C. (2011). Foundations of traversal based query execution over Linked Data (extended version). *CoRR*, abs/1108.6328.

Hartig, O. and Huber, F. (2011). A main memory index structure to query Linked Data. In *Linked Data on the web (LDOW2012)*.

Hartung, M., Kirsten, T., and Rahm, E. (2008). Analyzing the evolution of life science ontologies and mappings. In *Proceedings of the 5th international workshop on Data Integration in the Life Sciences*, DILS '08, pages 11–27, Berlin, Heidelberg. Springer-Verlag. Available from: `http://dx.doi.org/10.1007/978-3-540-69828-9_4`.

Hausenblas, M. and Karnstedt, M. (2010). Understanding Linked Open Data as a Web-scale database. In Laux, F. and Strömbäck, L., editors, *DBKDA*, pages 56–61. IEEE Computer Society.

Hayes, J. and Gutiérrez, C. (2004). Bipartite graphs as intermediate model for RDF. In [McIlraith et al., 2004], pages 47–61.

Hayes, P. and McBride, B. RDF Semantics [online]. (2004). Available from: `http://www.w3.org/TR/rdf-mt/`.

Heath, T. and Bizer, C. (2011). *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers.

Heimbigner, D. and McLeod, D. (1985). A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3(3):253–278.

Heine, F. (2006). Scalable P2P based RDF querying. In Jia, X., editor, *Infoscale*, volume 152 of *ACM International Conference Proceeding Series*, page 17. ACM.

Heine, F., Hovestadt, M., and Kao, O. (2005). Processing complex RDF queries over P2P networks. In *Proceedings of the 2005 ACM workshop on Information retrieval in peer-to-peer networks*, P2PIR '05, pages 41–48, New York, NY, USA. ACM. Available from: `http://doi.acm.org/10.1145/1096952.1096960`.

Henzinger, M. R., Heydon, A., Mitzenmacher, M., and Najork, M. (1999). Measuring index quality using random walks on the web. *Computer Networks*, 31(11-16):1291–1303.

Henzinger, M. R., Motwani, R., and Silverstein, C. (2002). Challenges in Web search engines. *SIGIR Forum*, 36(2):11–22.

Hogan, A. (2011). *Exploiting RDFS and OWL for Integrating Heterogeneous, Large-Scale, Linked Data Corpora*. PhD thesis, DERI, National University of Ireland, Galway.

Hogan, A., Harth, A., and Polleres, A. (2009). Scalable Authoritative OWL Reasoning for the web. *Int. J. Semantic Web Inf. Syst.*, 5(2):49–90.

Hogan, A., Harth, A., Umbrich, J., Kinsella, S., Polleres, A., and Decker, S. (2011). Searching and browsing Linked Data with SWSE: The Semantic Web Search Engine. *J. Web Sem.*, 9(4):365–401.

Hogan, A., Polleres, A., Umbrich, J., and Zimmermann, A. (2010). Some entities are more equal than others: statistical methods to consolidate Linked Data. In *Workshop on New Forms of Reasoning for the Semantic Web: Scalable & Dynamic (NeFoRS10)*.

Hogan, A., Umbrich, J., Harth, A., Cyganiak, R., Polleres, A., and Decker, S. (2012a). An empirical survey of Linked Data conformance. *J. Web Sem.*, 14.

Hogan, A., Zimmermann, A., Umbrich, J., Polleres, A., and Decker, S. (2012b). Scalable and distributed methods for entity matching, consolidation and disambiguation over Linked Data corpora. *J. Web Sem.*, 10:76–110.

Hose, K. (2009). *Processing Rank-Aware Queries in Schema-Based P2P Systems*. PhD thesis, TU Ilmenau.

Hose, K., Karnstedt, M., Koch, A., Sattler, K.-U., and Zinn, D. (2005). Processing rank-aware queries in P2P systems. In Moro, G., Bergamaschi, S., Joseph, S., Morin, J.-H., and Ouksel, A. M., editors, *DBISP2P*, volume 4125 of *Lecture Notes in Computer Science*, pages 171–178. Springer.

Hose, K., Klan, D., and Sattler, K.-U. (2006). Distributed data summaries for approximate query processing in PDMS. In *IDEAS*, pages 37–44. IEEE Computer Society.

Hose, K., Lemke, C., and Sattler, K.-U. (2009). Maintenance strategies for routing indexes. *Distributed and Parallel Databases*, 26(2-3):231–259.

Hose, K., Schenkel, R., Theobald, M., and Weikum, G. (2011). Database foundations for scalable RDF processing. In Polleres, A., d'Amato, C., Arenas, M., Handschuh, S., Kroner, P., Ossowski, S., and Patel-Schneider, P. F., editors, *Reasoning Web*, volume 6848 of *Lecture Notes in Computer Science*, pages 202–249. Springer.

Huang, S.-H. S. (1985). Multidimensional extendible hashing for partial-match queries. *International Journal of Parallel Programming*, 14(2):73–82.

Ioannidis, Y. E. (2003). The history of histograms (abridged). In *VLDB*, pages 19–30.

Isele, R., Umbrich, J., Bizer, C., and Harth, A. (2010). LDspider: An open-source crawling framework for the Web of Linked Data. In Polleres, A. and Chen, H., editors, *ISWC Posters&Demos*, volume 658 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Jacobs, I. and Walsh, N. Architecture of the World Wide Web, Volume One [online]. (2004) [cited 10-01-2011]. Available from: `http://www.w3.org/TR/webarch/`.

Jarke, M., Carey, M. J., Dittrich, K. R., Lochovsky, F. H., Loucopoulos, P., and Jeusfeld, M. A., editors (1997). *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann.

Käfer, T., Umbrich, J., Hogan, A., and Polleres, A. (2012). Towards a Dynamic Linked Data Observatory. In *Linked Data on the Web (LDOW)*.

Karnstedt, M. (2009). *Query Processing in a DHT-Based Universal Storage*. AVM-Verlag. PhD thesis.

Karnstedt, M., Sattler, K.-U., and Hauswirth, M. (2012). Scalable distributed indexing and query processing over Linked Data. *J. Web Sem.*, 10:3–32.

Karnstedt, M., Sattler, K.-U., Richtarsky, M., Müller, J., Hauswirth, M., Schmidt, R., and John, R. (2007). UniStore: Querying a DHT-based universal storage. In Chirkova, R., Dogac, A., Özsu, M. T., and Sellis, T. K., editors, *ICDE*, pages 1503–1504. IEEE.

Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632.

Ladwig, G. and Harth, A. (2011). CumulusRDF: Linked Data management on nested key-value stores. In *Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011) at the 10th International Semantic Web Conference (ISWC2011)*. -.

Ladwig, G. and Tran, T. (2010). Linked Data query processing strategies. In [Patel-Schneider et al., 2010], pages 453–469.

Ladwig, G. and Tran, T. (2011). SIHJoin: Querying remote and local Linked Data. In [Antoniou et al., 2011], pages 139–153.

Langegger, A. and Wöß, W. (2009). RDFStats - an extensible RDF statistics generator and library. In Tjoa, A. M. and Wagner, R., editors, *DEXA Workshops*, pages 79–83. IEEE Computer Society.

Langegger, A., Wöß, W., and Blöchl, M. (2008). A Semantic Web middleware for virtual data integration on the Web. In [Bechhofer et al., 2008], pages 493–507.

Lassila, O. and Swick, R. R. (1999). Resource Description Framework (RDF) model and syntax specification. W3c recommendation, W3C. Available from: `http://www.w3c.org/TR/REC-rdf-syntax`.

Lee, B. T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*.

Lee, H.-T., Leonard, D., Wang, X., and Loguinov, D. (2009). Irlbot: Scaling to 6 billion pages and beyond. *TWEB*, 3(3).

Li, Y. and Heflin, J. (2010). Using reformulation trees to optimize queries over distributed heterogeneous sources. In *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, pages 502–517.

Lim, L., Wang, M., Padmanabhan, S., Vitter, J. S., and Agarwal, R. C. (2001). Characterizing web document change. In Wang, X. S., Yu, G., and Lu, H., editors, *Advances in Web-Age Information Management (WAIM)*, volume 2118 of *Lecture Notes in Computer Science*, pages 133–144, Xi'an, China. Springer.

Maali, F., Cyganiak, R., and Peristeras, V. (2012). A publishing pipeline for linked government data. In [Simperl et al., 2012], pages 778–792.

Marzolla, M., Mordacchini, M., and Orlando, S. (2006). Tree vector indexes: Efficient range queries for dynamic content on peer-to-peer networks. In *PDP*, pages 457–464. IEEE Computer Society.

McIlraith, S. A., Plexousakis, D., and van Harmelen, F., editors (2004). *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, volume 3298 of *Lecture Notes in Computer Science*, Hiroshima, Japan. Springer.

Miller, L., Seaborne, A., and Reggiori, A. (2002). Three implementations of SquishQL, a simple RDF query language. In Horrocks, I. and Hendler, J. A., editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 423–435, Sardinia, Italy. Springer.

Morsey, M., Lehmann, J., Auer, S., and Ngomo, A.-C. N. (2011). DBpedia SPARQL benchmark - performance assessment with real queries on real data. In [Aroyo et al., 2011], pages 454–469.

Muñoz, S., Pérez, J., and Gutierrez, C. (2009). Simple and efficient minimal RDFS. *J. Web Sem.*, 7(3):220–234.

Muralikrishna, M. and DeWitt, D. J. (1988). Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In Boral, H. and Larson, P.-Å., editors, *SIGMOD Conference*, pages 28–36. ACM Press.

Nadeau, D. and Sekine, S. (2007). A survey of named entity recognition and classification. *Linguisticae Investigationes*, 30(1):3–26. Publisher: John Benjamins Publishing Company. Available from: http://www.ingentaconnect.com/content/jbp/li/2007/00000030/00000001/art00002.

Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmér, M., and Risch, T. (2002). EDUTELLA: a P2P networking infrastructure based on RDF. In *WWW*, pages 604–615.

Neumann, T. and Weikum, G. (2008). RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659.

Neumann, T. and Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113.

Nierstrasz, O. W3Catalog [online]. (1996). Available from: http://scg.unibe.ch/archive/software/w3catalog/.

Ntoulas, A., Cho, J., and Olston, C. (2004). What's new on the web?: the evolution of the web from a search engine perspective. In [Feldman et al., 2004], pages 1–12.

Oita, M. and Senellart, P. (2011). Deriving Dynamics of Web Pages: A Survey. In *TWAW (Temporal Workshop on Web Archiving)*, Hyderabad, Inde. Available from: http://hal.inria.fr/inria-00588715.

Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., and Tummarello, G. (2008). Sindice.com: a document-oriented lookup index for open Linked Data. *IJMSO*, 3(1):37–52.

O'Riain, S., Harth, A., and Curry, E. (2012). Linked Data driven information systems as an enabler for integrating financial data. In *Information Systems for Global Financial Markets: Emerging Developments and Effects*, chapter 10, pages 239–270. IGI Global.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the Web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120. Available from: http://ilpubs.stanford.edu:8090/422/.

Pandey, S., Ramamritham, K., and Chakrabarti, S. (2003). Monitoring the dynamic Web to respond to continuous queries. In *WWW*, pages 659–668.

Patel-Schneider, P. F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J. Z., Horrocks, I., and Glimm, B., editors (2010). *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, volume 6496 of *Lecture Notes in Computer Science*, Shanghai, China. Springer.

Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3).

Pérez, J., Arenas, M., and Gutierrez, C. (2010). nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270.

Petrakis, Y., Koloniari, G., and Pitoura, E. (2004). On using histograms as routing indexes in peer-to-peer systems. In Ng, W. S., Ooi, B. C., Ouksel, A. M., and Sartori, C., editors, *DBISP2P*, volume 3367 of *Lecture Notes in Computer Science*, pages 16–30. Springer.

Petrakis, Y. and Pitoura, E. (2004). On constructing small worlds in unstructured peer-to-peer systems. In Lindner, W., Mesiti, M., Türker, C., Tzitzikas, Y., and Vakali, A., editors, *EDBT Workshops*, volume 3268 of *Lecture Notes in Computer Science*, pages 415–424. Springer.

Pollock, R., Keegan, M., and Walsh, J. Comprehensive Knowledge Archive Network [online]. (2004). Available from: `http://ckan.org/`.

Poosala, V. and Ioannidis, Y. E. (1997). Selectivity estimation without the attribute value independence assumption. In [Jarke et al., 1997], pages 486–495.

Popitsch, N. and Haslhofer, B. (2011). Dsnotify - a solution for event detection and link maintenance in dynamic datasets. *J. Web Sem.*, 9(3):266–283.

Press, A., editor (1999). *Patterns of Search: Analyzing and Modeling Web Query Refinement.* Available from: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.8813`.

Prud'hommeaux, E. and Seaborne, A. SPARQL query language for RDF [online]. (2008). Available from: `http://www.w3.org/TR/rdf-sparql-query/`.

Quilitz, B. and Leser, U. (2008). Querying distributed RDF data sources with SPARQL. In [Bechhofer et al., 2008], pages 524–538.

Raskino, M., Fenn, J., and Linden, A. (2005). Extracting value from the massively connected world of 2015. Technical Report Technical Report G00125949, Gartner Research Gartner Research.

Rathi, A., Lu, H., and Hedrick, G. E. (1990). Performance comparison of extendible hashing and linear hashing techniques. In *SIGSMALL/PC Symposium*, pages 178–185.

RSS. Really simple syndication [online]. (1999). Available from: `http://www.rss.com/`.

Schlosser, M. T., Sintek, M., Decker, S., and Nejdl, W. (2002). Hypercup - hypercubes, ontologies, and efficient search on peer-to-peer networks. In Moro, G. and Koubarakis, M., editors, *AP2PC*, volume 2530 of *Lecture Notes in Computer Science*, pages 112–124. Springer.

Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., and Tran, T. (2011). Fedbench: A benchmark suite for federated semantic data query processing. In [Aroyo et al., 2011], pages 585–600.

Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., and Pinkel, C. (2008a). An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In [Sheth et al., 2008], pages 82–97.

Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2008b). Sp2bench: A SPARQL performance benchmark. *CoRR*, abs/0806.4627.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: A federation layer for distributed query processing on Linked Open Data. In [Antoniou et al., 2011], pages 481–486.

Senellart, P., Abiteboul, S., and Gilleron, R. (2008). Understanding the hidden web. *ERCIM News*, 2008(72).

Sheth, A. P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T. W., and Thirunarayan, K., editors (2008). *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, volume 5318 of *Lecture Notes in Computer Science*, Karlsruhe, Germany. Springer.

Simperl, E., Cimiano, P., Polleres, A., Corcho, Ó., and Presutti, V., editors (2012). *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, volume 7295 of *Lecture Notes in Computer Science*, Crete, Greece. Springer.

Srivastava, U., Haas, P. J., Markl, V., Kutsch, M., and Tran, T. M. (2006). Isomer: Consistent histogram construction using query feedback. In Liu, L., Reuter, A., Whang, K.-Y., and Zhang, J., editors, *ICDE*, page 39. IEEE Computer Society.

Stocker, M. and Seaborne, A. (2007). Arqo: The architecture for an arq static query optimizer. Technical report, HP Labs Bristol.

Stuckenschmidt, H., Vdovjak, R., Broekstra, J., and Houben, G.-J. (2005). Towards distributed processing of RDF path queries. *Int. J. Web Eng. Technol.*, 2(2/3):207–230.

Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., and Broekstra, J. (2004). Index structures and algorithms for querying distributed RDF repositories. In [Feldman et al., 2004], pages 631–639.

Suchanek, F. M., Kasneci, G., and Weikum, G. (2008). Yago: A large ontology from wikipedia and wordnet. *J. Web Sem.*, 6(3):203–217.

Swartz, A. application/rdf+xml media type registration [online]. (2004). Available from: `http://tools.ietf.org/html/rfc3870`.

Tian, Y., Umbrich, J., and Yu, Y. (2011). Enhancing source selection for live queries over Linked Data via query log mining. In *JIST*, volume 7185 of *Lecture Notes in Computer Science*, pages 176–191. Springer.

Toffler, A. (1980). *The third wave*. Morrow.

Tran, T., Zhang, L., and Studer, R. (2010). Summary models for routing keywords to Linked Data sources. In [Patel-Schneider et al., 2010], pages 781–797.

Tummarello, G., Morbidoni, C., Bachmann-Gmür, R., and Erling, O. (2007). RDFSync: Efficient remote synchronization of RDF models. In [Aberer et al., 2007], pages 537–551.

Umbrich, J., Hausenblas, M., Hogan, A., Polleres, A., and Decker, S. (2010a). Towards dataset dynamics: Change frequency of Linked Open Data sources. In Bizer, C., Heath, T., Berners-Lee, T., and Hausenblas, M., editors, *LDOW*, volume 628 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Umbrich, J., Hogan, A., Polleres, A., and Decker, S. (2012a). Improving the recall of live Linked Data querying through reasoning. In *International Conference on Web Reasoning and Rule Systems*, Vienna, Austria.

Umbrich, J., Hose, K., Karnstedt, M., Harth, A., and Polleres, A. (2011). Comparing data summaries for processing live queries over Linked Data. *World Wide Web*, 14(5-6).

Umbrich, J., Karnstedt, M., Hogan, A., and Parreira, J. X. (2012b). Freshening up while staying fast: Towards hybrid SPARQL queries. In *Knowledge Engineering and Knowledge Management*, Galway, Ireland.

Umbrich, J., Karnstedt, M., Hogan, A., and Parreira, J. X. (2012c). Hybrid SPARQL query processing: fresh vs. fast results. In *International Semantic Web Conference*, Boston, USA.

Umbrich, J., Karnstedt, M., and Land., S. (2010b). Towards understanding the changing web: Mining the dynamics of linked-data sources and entities. In Atzmüller, M., Benz, D., Hotho, A., and Stumme, G., editors, *Proceedings of LWA2010 - Workshop-Woche: Lernen, Wissen & Adaptivitaet*, Kassel, Germany.

Umbrich, J., Karnstedt, M., Parreira, J. X., Polleres, A., and Hauswirth, M. (2012d). Linked Data and live querying for enabling support platforms for Web dataspaces. In *Proceedings of the 3rd International Workshop on Data Engineering Meets the Semantic Web (DESWeb), co-located with ICDE2012*.

Umbrich, J., Villazon-Terrazas, B., and Hausenblas, M. (2010c). Dataset dynamics compendium: Where we are so far! In *Consuming Linked Data Workshop (COLD) at ISWC*, Shanghai.

Vose, D. and Vose, D. (2000). *Risk analysis : a quantitative guide*. Wiley. Available from: `http://www.worldcat.org/isbn/9780471997658`.

Vrandečić, D., Krötzsch, M., Rudolph, S., and Lösch, U. (2010). Leveraging nonlexical knowledge for the Linked Open Data Web. *Review of April Fool's day Transactions*, pages 18–27. Available from: `http://km.aifb.kit.edu/projects/numbers/linked_open_numbers.pdf`.

Weiss, C., Karras, P., and Bernstein, A. (2008). Hexastore: sextuple indexing for Semantic Web data management. *PVLDB*, 1(1):1008–1019.

Wildemuth, B. M. (2004). The effects of domain knowledge on search tactic formulation. *JASIST*, 55(3):246–258.

Williams, G. T. SPARQL 1.1 service description [online]. (2012). Available from: `http://www.w3.org/TR/sparql11-service-description/`.

Williams, G. T. and Weaver, J. (2011). Enabling fine-grained HTTP caching of SPARQL query results. In [Aroyo et al., 2011], pages 762–777.

Zinn, D. (2004). Skyline queries in P2P systems. Master's thesis, TU Ilmenau.

PREFIXES

A

Prefix	URI
cb:	http://www.bizer.de#
cbDoc:	http://www4.wiwiss.fu-berlin.de/bizer/foaf.rdf
dblpA:	http://dblp.l3s.de/d2r/resource/authors/
dblpADoc:	http://dblp.l3s.de/d2r/data/authors/
dblpP:	http://dblp.l3s.de/d2r/resource/publications/conf/semweb/
dblpPDoc:	http://dblp.l3s.de/d2r/data/publications/conf/semweb/
dbpcat:	http://dbpedia.org/resource/Category:
dbpedia:	http://dbpedia.org/resource/
dbpprop:	http://dbpedia.org/property/
dbpowl:	http://dbpedia.org/ontology/
dcterms:	http://purl.org/dc/terms/
drugbank:	http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/
ebiz:	http://www.ebusiness-unibw.org/ontologies/consumerelectronics/v1
foaf:	http://xmlns.com/foaf/0.1/
geo:	http://www.geonames.org/ontology#
nytimes:	http://data.nytimes.com/elements/
oh:	http://www.informatik.hu-berlin.de/~hartig/foaf.rdf#
ohDoc:	http://www.informatik.hu-berlin.de/~hartig/foaf.rdf
owl:	http://www.w3.org/2002/07/owl#
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
skos:	http://www.w3.org/2004/02/skos/core#
swc:	http://data.semanticweb.org/ns/swc/ontology#
swrc:	http://swrc.ontoware.org/ontology#
swIswc08pd:	http://data.semanticweb.org/conference/iswc/2008/poster_demo_proceedings
swEswc10:	http://data.semanticweb.org/conference/eswc/2010

Table A.1: Mappings for all prefixes used

B ACCESS AND LIFESPAN OF SOURCES

Accesses [weeks] №	Lifespan [weeks] 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	TOTAL
1	14.47	–	–	–	–	–	–	–	–	–	–	–	–	–	–	14.47
2	–	1.75	0.67	0.18	0.22	0.41	0.13	0.17	0.06	0.16	0.64	0.02	0.03	0.03	0.03	4.51
3	–	–	0.99	0.64	0.26	0.41	0.26	0.25	0.09	0.24	0.38	0.05	0.05	0.09	0.09	3.79
4	–	–	–	1.06	0.47	1.2	0.38	0.12	0.21	0.14	0.43	0.1	0.07	0.17	0.04	4.41
5	–	–	–	–	1.17	0.22	0.1	0.21	0.09	0.13	0.21	0.63	0.08	0.49	0.08	3.42
6	–	–	–	–	–	0.82	0.18	0.6	0.05	0.08	0.12	0.27	0.82	0.29	0.13	3.36
7	–	–	–	–	–	–	0.62	0.5	0.15	0.06	0.13	0.32	0.36	0.57	0.21	2.93
8	–	–	–	–	–	–	–	0.87	0.2	0.07	0.08	0.2	0.25	0.59	0.4	2.66
9	–	–	–	–	–	–	–	–	2.18	0.09	0.09	0.14	0.29	0.49	0.84	4.11
10	–	–	–	–	–	–	–	–	–	0.26	0.15	0.24	0.14	0.47	0.79	2.04
11	–	–	–	–	–	–	–	–	–	–	0.25	0.36	0.21	1.12	1.08	3.02
12	–	–	–	–	–	–	–	–	–	–	–	0.83	0.25	1.36	1.7	4.13
13	–	–	–	–	–	–	–	–	–	–	–	–	0.56	1.68	4.06	6.3
14	–	–	–	–	–	–	–	–	–	–	–	–	–	1.78	9.59	11.37
15	–	–	–	–	–	–	–	–	–	–	–	–	–	–	29.48	29.48
	14.47	1.75	1.66	1.88	2.13	3.07	1.67	2.72	3.03	1.23	2.48	3.15	3.11	9.13	48.51	100

Table B.1: Fraction of sources with certain access and lifespan combination (presented as percentage).

C

FEDBENCH QUERIES

Herein, we present the results for the individual FedBench queries. We run the queries four times for each of the ten LiDaQ experiments, and for comparability across different configurations, we present the best run in terms of results returned, and if tied, by time; we thus select the run which provided the most stable behaviour and returned the most results. The variation between the four runs has already been analysed in Section 4.5.2.1. We also show results for the SQUIN library: we highlight that we only run the SQUIN implementation once since (to the best of our knowledge) it does not implement politeness policies, and thus the LiDaQ configurations may have an advantage in comparison—in any case, we show that SQUIN is generally faster. Note that we do not have measurements for the triples processed by SQUIN.

To avoid repetition, we discuss results incrementally; we may only briefly remark again on observations that have already been made for earlier queries.

```
SELECT DISTINCT *
WHERE {
  ?paper swc:isPartOf swIswc08pd: .
  ?paper swrc:author ?p .
  ?p rdfs:label ?n .
}
```

Query C.1: [LD1]: List author(s) with their paper(s) for the poster/demo track of ISWC 2008.

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
Core	582	333	342.9	4.9	633	23,968	–
Core$^-$	582	333	343.5	3.7	628	23,013	–
seeAlso	582	333	361.5	3.6	704	25,229	–
sameAs	668	529	391.4	3.9	761	26,269	8,109
RDFS$_s$	615	380	478.8	3.8	628	23,013	11,501
RDFS$_d$	615	380	350.3	5.2	666	23,013	13,984
RDFS$_e$	615	380	356.1	6.6	865	23,013	18,587
Comb$_s$	715	692	571.9	4.4	842	29,212	26,804
Comb$_d$	713	680	461	8.4	1,002	28,485	27,745
Comb$_e$	715	692	512.6	15.8	1,269	29,212	35,418
Squin	582	333	86.8	28	703	–	–

Table C.1: Benchmark results for query LD1

The results for Query C.1 come mostly from one site: the `data.semanticweb.org` "Dog Food" server. The query engine first finds the list of URIs for all 85 demo/-poster papers published at ISWC 2008 on the first document, dereferences these 85 URIs and builds a list of 288 unique authors, then finally dereferences these to find a list of 333 unique names (some authors have multiple versions of names, particularly for abbreviations). As such, we see that the overall time taken for baseline

LiDaQ methods is roughly a function of the politeness policy (two lookups per second) and the number of HTTP lookups required: the query times of over 5 minutes are attributable to the high number of sources that must be accessed. Conversely, although Squin performs more lookups than, *e.g.*, Core$^-$ and Core, and generates the same results, it is much faster, but only by performing at least eight HTTP lookups per second to `data.semanticweb.org` which is four times more than the bounds of our politeness policy.

In this case, we see that Core$^-$ saves few lookups and little time when compared with Core, and that seeAlso increases the number of sources but not the number of results. We see that RDFS reasoning finds some additional results: `foaf:name` and `skos:prefLabel` are found to be sub-properties of `rdfs:label` and provide additional name variations, including with language tags. Some of the authors have `owl:sameAs` relations to external sources, which, with sameAs, provide additional URIs for authors and name variations using a sub-property of label. The most results are thus given by the Comb approaches, which are also the slowest overall.[1]

```
SELECT DISTINCT *
WHERE {
  ?proceedings swc:relatedToEvent swEswc10: .
  ?paper swc:isPartOf ?proceedings .
  ?paper swrc:author ?p .
}
```

Query C.2: [LD2]: List author(s) with their paper(s) in proceedings related to ESWC 2010.

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
Core	236	185	260.3	3.9	478	20,356	–
Core$^-$	236	185	69.8	3.5	128	3,662	–
seeAlso	236	185	70	3.5	128	3,662	–
sameAs	236	185	70.3	3.6	128	3,662	–
RDFS$_s$	236	185	202.5	4	128	3,662	2,139
RDFS$_d$	236	185	73.6	5.7	148	3,662	8,193
RDFS$_e$	236	185	77.7	7	363	3,662	12,312
Comb$_s$	236	185	219	4.8	128	3,662	2,139
Comb$_d$	236	185	76.9	12.8	148	3,662	8,124
Comb$_e$	236	185	79.7	22.9	363	3,662	12,162
Squin	236	185	24	4.4	171	–	–

Table C.2: Benchmark results for query LD2

Although Query C.2 is very similar to Query C.1—requiring data mostly from the same Dog-Food provider—the measures in Table C.2 tell a different story. The results are the same for all configurations: none of the extensions find any additional results in this case, though they do add an additional 10 seconds to the results. In fact, given that sameAs and RDFS$_s$ retrieve the same number of sources as Core$^-$, this result gives us an insight into the local overhead of reasoning, which we see has little effect on query times.

The most striking observation is the source-selection savings for Core$^-$ vs. Core, where Core does not dereference the 173 authors bound to `?p` (requiring

1 The additional answers available for RDFS and same-as reasoning can be seen from, *e.g.*, `http://data.semanticweb.org/person/mathieu-daquin/rdf`.

$173 \times 2 = 346$ lookups including `303` redirects) since `?p` bindings are not part of a join, translating into a major time saving. We also note that SQUIN performs fewer lookups than we would expect if it were to dereference authors, but still dereferences more URIs than CORE$^-$ and its analogues. As such, it would seem that SQUIN also implements some reduced source-selection optimisations.

```
SELECT DISTINCT *
WHERE {
  ?paper swc:isPartOf swIswc08pd: .
  ?paper swrc:author ?p .
  ?p owl:sameAs ?x ; rdfs:label ?n .
}
```

Query C.3: [LD3]: List the author(s) with their same-as relation(s), and with their paper(s) for the poster/demo track of ISWC 2008.

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
CORE	247	191	388.1	4	760	27,538	–
CORE$^-$	247	191	342.8	8.1	628	23,013	–
SEEALSO	247	191	360	8.2	704	25,229	–
SAMEAS	394	951	389.5	4.2	763	26,248	8,014
RDFS$_s$	263	246	474.7	5	628	23,013	11,501
RDFS$_d$	263	246	349.5	4.9	666	23,013	13,991
RDFS$_e$	263	246	355.8	4.8	865	23,013	18,775
COMB$_s$	422	1,469	569.3	10.4	839	28,485	25,797
COMB$_d$	425	1,583	461.8	18.8	1,008	29,212	28,814
COMB$_e$	425	1,583	511.6	19.1	1,269	29,212	35,547
SQUIN	247	191	87.3	32.2	728	–	–

Table C.3: Benchmark results for query LD3

Query C.3 adds a triple pattern to query Query C.1, restricting the list of authors to (explicitly) look for those with an `owl:sameAs` relation. This reduces the number of authors involved to 288 in Query C.1 to 54 in Query C.3. We can see in Table C.3 that for configurations without reasoning, Query C.3 returns ~57% of the number of results of Query C.1: the decrease in authors is partially balanced by the addition of another variable in the results. CORE$^-$ offers a moderate performance improvement over CORE while returning the same results. RDFS reasoning increases result sizes for similar reasons as before, and at little cost. SAMEAS shows a marked increase in results size: the additional `?x` variable is replaced by all equivalent URIs for each author, leading to an additional product of result terms.

```
SELECT DISTINCT * WHERE {
  ?role swc:isRoleAt swEswc10: .
  ?role swc:heldBy ?p .
  ?paper swrc:author ?p .
  ?paper swc:isPartOf ?proceedings .
  ?proceedings swc:relatedToEvent swEswc10: .
}
```

Query C.4: [LD4]: List the author(s) with paper(s) in the proceedings of ESWC 2010 who also had role(s) at the conference.

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
Core	60	50	986.9	33.5	1,805	74,635	–
Core$^-$	60	50	984.5	50.9	1,801	73,767	–
seeAlso	60	50	1,019.4	51.9	1,982	81,864	–
sameAs	105	146	1,167.4	56.5	2,462	102,286	104,431
RDFS$_s$	60	50	1,140.2	17.5	1,801	73,767	45,352
RDFS$_d$	60	50	1,003	66	1,843	73,767	45,943
RDFS$_e$	60	50	1,023.2	11.5	2,173	73,767	58,374
Comb$_s$	162	203	4,658.4	68.9	2,834	115,707	297,620
Comb$_d$	162	203	2,249.4	172.4	3,383	115,663	557,880
Comb$_e$	80	126	7,211.6	52.9	9,225	109,858	1,702,602
Squin	60	50	244.3	236.7	1,981	–	–

Table C.4: Benchmark results for query LD4

Again, Query C.4 is an extension of Query C.2 and restricts the list of authors to those who, as well as having a paper at ESWC 2010, also had a role at the conference. Looking at the results in Table C.4, even for Core$^-$, the query processor performed over 1,800 lookups and our source selection approach does not affect the number of lookups (in this case, `?p` falls into a join position and 251 people had a role at ISWC). The fastest time was around 16 minutes for Core$^-$ (again, approximately $\frac{1,800}{2}$ seconds). RDFS reasoning alone produces no additional results, but also does not overly influence runtime. Conversely, sameAs produces additional results, where author pages are this time dereferenced and `owl:sameAs` relations found, adding aliases for bindings in `?p`. The combined approaches became unstable, adding additional HTTP load to what is already a demanding query. In particular, Comb$_s$ and Comb$_e$ actually timeout after roughly two hours; for example, the Comb$_e$ approach retrieved almost ten thousand sources before timing out, where the `owl:sameAs` links from authors on `data.semanticweb.org` form a bridge to DBpedia, whose schema data has a high fan-out. From previous queries, we have seen that the schema data directly referenced from `data.semanticweb.org` is relatively easy to retrieve using dynamic import mechanisms; however, the schemata for other sites requires many more sources to retrieve, particularly in the RDFS$_e$/Comb$_e$ configurations.

```
SELECT DISTINCT *
WHERE {
  ?a dbowl:artist dbpedia:Michael_Jackson .
  ?a rdf:type dbowl:Album .
  ?a foaf:name ?n .
}
```

Query C.5: [LD5]: List the name(s) of the album(s) by Michael Jackson.

This query shifts the focus to the `dbpedia.org` data provider. First `dbpedia:Michael_Jackson` is dereferenced to retrieve URIs for Michael Jackson's albums, which are subsequently dereferenced to confirm that they are albums and to retrieve their name. Primarily, the results show that following `owl:sameAs` links from the DBpedia domain introduces high overhead: there are a total of 425 URI aliases for Michael Jackson and his albums on the DBpedia, including `owl:sameAs` links to `freebase.com`, `sw.cyc.com`, `linkedmdb.org` and `zitgist.com`. Although the

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
Core	85	43	63.3	6	121	9,017	–
Core$^-$	85	43	66.8	9	116	8,285	–
seeAlso	85	43	65.2	7.2	116	8,285	–
sameAs	313	271	212.7	14	593	15,179	119,907
RDFS$_s$	85	43	218.3	23.5	116	8,284	7,115
RDFS$_d$	83	42	417.7	9.4	698	8,203	163,163
RDFS$_e$	36	18	4,886.6	12.9	9,729	4,087	302,609
Comb$_s$	0	0	7,341.5	–	780	17,027	745,534
Comb$_d$	15	14	7,200.6	353.7	1,452	14,450	958,611
Comb$_e$	–	–	–	–	–	–	–
Squin	85	43	16.2	3.9	115	–	–

Table C.5: Benchmark results for query LD5

sameAs configuration runs through (taking 3.28× longer than Core$^-$), when same-as and RDFS reasoning are combined, LiDaQ becomes unstable: all of the Comb approaches timed out, where Comb$_e$ threw an `OutOfMemoryException` in all four runs before the timeout was reached due to massive amounts of inferences. Furthermore, the RDFS$_e$ configuration without `owl:sameAs` reasoning showed that the dynamic import of extended schema does not work well for DBpedia, again touching upon nearly ten thousand sources and generating fewer results than Core$^-$ (which it extends). In general, the high fan-out of `owl:sameAs` and schema-level links on DBpedia—and on sites linked by DBpedia such as `sw.cyc.com`—combined with a query that already accesses over one hundred DBpedia pages in the baseline setup, prove too much for RDFS$_e$ and Comb approaches.

```
SELECT DISTINCT * WHERE {
  ?director dbowl:nationality dbpedia:Italy .
  ?film dbowl:director ?director.
  ?x owl:sameAs ?film .
  ?x foaf:based_near ?y .
  ?y geo:officialName ?n .
}
```

Query C.6: [LD6]: List the movie director(s) from Italy, their film(s) and the official name(s) of location(s) for the film(s).

Query C.6 intends to span the DBpedia (first three patterns), LinkedMDB (fourth pattern) and GeoNames (fifth pattern) data providers. However, as we can see in Table C.6, no setup returned any results. At the time of running the experiments, the dereferenced document for `dbpedia:Italy` contained 10,001 triples due to a manual cut-off set for the exporter [2], where many triples (including inlinks) were omitted and where the dereferenced document included no `dbowl:nationality` triples. At the time of writing, the dereferenced document contains 44,421 triples, including 842 `dbowl:nationality` inlinks. [3] In any case, as we discuss for the next query, the GeoNames exporter hosting data for the final triple pattern bans access from all agents through its `robots.txt`. Aside from such issues, we would expect this query to offer a major challenge to LTBQE, and to again introduce unstable behaviour for Comb configurations.

2 Last accessed on 2012/02/28.
3 Last accessed on 2012/08/09.

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
Core	0	0	9.6	–	12	17,864	–
Core$^-$	0	0	6.8	–	2	10,001	–
seeAlso	0	0	3.6	–	2	10,001	–
sameAs	0	0	49.5	–	7	10,067	20,090
RDFS$_s$	0	0	145.2	–	2	10,001	4,580
RDFS$_d$	0	0	27.3	–	49	10,001	1,329
Comb$_s$	0	0	215.3	–	7	10,067	24,922
Comb$_d$	0	0	59	–	64	10,067	71,560
Comb$_e$	0	0	7,223.5	–	945	10,067	427,421
Squin	0	0	5.9	–	1	–	–

Table C.6: Benchmark results for query LD6

```
SELECT DISTINCT *
WHERE {
  ?x geo:parentFeature <http://sws.geonames.org/2921044/> .
  ?x geo:name ?n .
}
```

Query C.7: [LD7]: List the name(s) of the parent feature(s) of Germany.

LiDaQ will not run Query C.7 since the `robots.txt`[4] forbids software agents to access information on the `sws.geonames.org` domain. SQUIN does access the `sws.geonames.org` domain, but even aside from the `robots.txt` issue, the first query pattern is not matched by any data in the document dereferenced by the given GeoNames URI for Germany: dereferenced documents on the GeoNames domain only contain triples where the URI in question appears in the subject position, not "inlinks". If not for these two issues, we would expect this query to be straightforward for LiDaQ/SQUIN to run.

```
SELECT DISTINCT *
WHERE {
   ?drug drugbank:drugCategory drugbank:micronutrient .
   ?drug drugbank:casRegistryNumber ?id .
   ?drug owl:sameAs ?s .
   ?s foaf:name ?o .
   ?s dcterms:subject ?sub .
}
```

Query C.8: [LD8]: List the drug(s) in the micronutrient category, their CAS registry number(s), alias(es), name(s) and subject(s).

From the results in Table C.7, we see the improvements of Core vs. Core$^-$ for Query C.8. Most prominently however, the results for Query C.8 show highly unstable behaviour for all reasoning extensions except RDFS$_s$. In particular, the consideration of `owl:sameAs` links snowballs and introduces massive problems, which we believe to be due to data quality issues with this relation within Linked Drug Data, and which we had previously observed in other work [Hogan et al., 2012b].[5] This of course highlights the problem whereby – even with counter measures such as authoritative analysis of schema data – reasoning exacerbates data

4 `http://sws.geonames.org/robots.txt`

5 We refer the reader to `https://groups.google.com/forum/?fromgroups#!topic/pedantic-web/rXQPcFLMOi0` for detailed discussion.

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
CORE	39	19	78.9	17.6	351	20,655	—
CORE$^-$	39	19	61.9	25.5	257	7,245	—
SEEALSO	39	19	96.2	21.4	334	7,309	—
SAMEAS	294	21,071	1,374.4	67.1	856	12,839	515,297
RDFS$_s$	39	19	198.5	15.9	257	7,245	4,979
RDFS$_d$	8	4	181.4	132.6	231	774	43,814
RDFS$_e$	24	12	7,514.8	155.5	7,175	5,491	143,236
COMB$_s$	347	29,139	7,354.9	35.5	1,217	15,209	407,741
COMB$_d$	0	0	7,212.1	—	1,289	11,416	73,304
COMB$_e$	—	—	—	—	—	—	—
SQUIN	22	10	120.9	10.5	482	—	—

Table C.7: Benchmark results for query LD8

quality issues for remote data providers. When `owl:sameAs` and dynamic RDFS import and reasoning is combined for COMB$_d$ and COMB$_e$, we encountered further `OutOfMemoryExceptions`.

```
SELECT DISTINCT *
WHERE {
  ?x dcterms:subject
      dbpcat:FIFA_World_Cup-winning_countries .
  ?p dbpowl:managerClub ?x .
  ?p foaf:name "Luiz Felipe Scolari"@en .
}
```

Query C.9: [LD9]: List the football team(s) that won a FIFA World Cup and that were managed by "Luiz Felipe Scolari".

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
CORE	0	0	147.3	—	266	29,326	—
CORE$^-$	0	0	147.4	—	260	27,821	—
SEEALSO	0	0	136.4	—	260	27,821	—
SAMEAS	0	0	182.6	—	337	7,915	92,485
RDFS$_s$	0	0	299.2	—	202	22,791	19,642
RDFS$_d$	0	0	904.8	—	1,512	19,133	227,663
RDFS$_e$	0	0	1,607.5	—	3,488	4,928	33,810
COMB$_s$	0	0	7,342.9	—	984	28,211	558,187
COMB$_d$	0	0	7,202	—	1,437	16,109	1,432,297
COMB$_e$	—	—	—	—	—	—	—
SQUIN	0	0	25.6	—	247	—	—

Table C.8: Benchmark results for query LD9

As we can see from the results in Table C.8, none of the setups returned any content for Query C.9. At the time of the experiments, the document for the Brazilian national football team (the answer to `?p`) contained parser errors, which have since

been fixed.[6] We again see that following `owl:sameAs` and schema level links from the DBpedia causes huge overheads, with COMB$_s$ and COMB$_d$ hitting timeouts, and COMB$_e$ again throwing an `OutOfMemoryException` after inferring too much data.

```
SELECT DISTINCT *
WHERE {
  ?n dcterms:subject dbpcat:Chancellors_of_Germany .
  ?n owl:sameAs ?p2 .
  ?p2 nytimes:latest_use ?u .
}
```

Query C.10: [LD10]: List the chancellor(s) of Germany, their alias(es) and latest article(s).

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
CORE	0	0	55.5	—	165	15,008	—
CORE$^-$	0	0	52.7	—	160	13,692	—
SEEALSO	0	0	52.6	—	160	13,692	—
SAMEAS	200	5,825	7,200.6	310.4	937	18,120	811,104
RDFS$_s$	0	0	195	—	160	13,692	17,008
RDFS$_d$	0	0	321	—	855	4,011	79,345
COMB$_s$	0	0	7,351	—	897	18,404	387,428
COMB$_d$	—	—	—	—	—	—	—
COMB$_e$	—	—	—	—	—	—	—
SQUIN	0	0	50.1	—	158	—	—

Table C.9: Benchmark results for query LD10

Query C.10 aims to combine data from the `dbpedia.org` domains and the `data.nytimes.com` domain. We already explained that we changed the query predicate `skos:subject` to `dcterms:subject` in order to reflect changes in the DBpedia data model. However, we found that although the content returned for the entities that are in the DBpedia category "Chancellors of Germany" contains several `owl:sameAs` relations to aliases in the `data.nytimes.com` domain, these are found in the inverse order of the query pattern. This fact is reflected in the results shown in Table C.9, where we only find results if `owl:sameAs` inferencing is enabled; in fact, both configurations which returned results timed out doing so, and again, both configurations involving the dynamic import of schema data threw exceptions.

```
SELECT DISTINCT *
WHERE {
  ?x dbpowl:team dbpedia:Eintracht_Frankfurt .
  ?x rdfs:label ?y .
  ?x dbpowl:birthDate ?d .
  ?x dbpowl:birthPlace ?p .
  ?p rdfs:label ?l .
}
```

Query C.11: [LD11]: List the name(s) of the player(s) on the Eintracht Frankfurt team, their birthday(s) and the name(s) of their birthplace(s).

6 The document URL in question—`http://dbpedia.org/data/Brazil_national_football_team.xml`—was tested with the W3C RDF/XML validator.

Setup	Terms	Results	Time (s)	First (s)	HTTP	Data	Inferred
Core	4,240	25,445	607.3	15.1	1,125	354,880	—
Core^{-}	3,936	23,621	595.8	7.6	1,073	336,615	—
seeAlso	4,194	25,068	599.2	12.7	1,113	353,961	—
sameAs	40	572	7,210.2	80.9	3,741	94,263	2,327,965
RDFS$_s$	1,496	6,982	8,297.4	25.6	450	128,281	103,769
RDFS$_d$	1,281	7,319	2,392.5	27.7	3,896	123,839	271,772
RDFS$_e$	—	—	—	—	—	—	—
Comb$_s$	259	77,484	7,345.9	104.5	3,077	97,925	575,314
Comb$_d$	275	196,448	7,201.6	51	5,974	92,285	1,577,530
Comb$_e$	240	157,198	7,207.9	92.2	17,996	21,574	1,930,660
Squin	2,673	15,900	158.9	31.1	1,116	—	—

Table C.10: Benchmark results for query LD11

Table C.10 shows that this query involves the largest amount of results and source lookups of all the FedBench queries.[7] The combination of over 300 players, each of which typically has labels in several languages and has two or three birth-places, each of which in turn has labels in several languages, leads to large results sets, even without reasoning. The number of HTTP lookups also reflects the breadth of this query, primarily due to lookups on players and places. The reasoning extensions again exhibit unstable behaviour, either eventually timing-out or throwing an exception.

7 We remark that the public centralised SPARQL endpoint for DBpedia often times-out with a `509` response code for this query.

D

QWALK RESULTS

With respect to the detailed average measures for the QWalk experiments, Table D.1 presents the results for entity-* queries, Table D.2 and Table D.4 gives results for star-* queries, and Table D.3 and Table D.5gives results for *-path-* queries. For space reasons, we only present standard deviations for terms, results and time.

	Setup	Term		Results		Time (s)		First (s)	HTTP	Data	Inferred
		avg.	σ	*avg.*	σ	*avg.*	σ				
entity-s	Core	19.35	±37.75	16.78	±37.94	16.79	±8.19	6.6	17.78	8,676.43	–
	Core⁻	19.33	±37.72	16.77	±37.91	9.99	±4.81	6.6	2.92	5,772.5	–
	seeAlso	19.33	±37.72	16.77	±37.91	10.04	±5.05	6.42	3.15	5,778.68	–
	sameAs	25.28	±63.42	22.73	±63.76	10.57	±5.47	6.93	3.27	5,579.42	67.82
	$RDFS_s$	24.13	±37.95	21.77	±38.01	16.93	±35.6	6.57	4.55	22,591.2	16,328.55
	$Comb_s$	30.07	±63.66	27.73	±63.91	18.08	±38.58	9.5	5.07	22,424.57	16,423.52
entity-o	Core	54.16	±382.48	53.53	±382.42	13.49	±8.5	6.19	7.18	3,517.61	–
	Core⁻	54.18	±382.47	53.51	±382.42	8.25	±5.72	6.17	2.61	1,284.07	–
	seeAlso	54.19	±382.47	53.53	±382.42	8.69	±5.65	6.16	2.77	1,832.04	–
	sameAs	55.04	±382.37	54.35	±382.32	9.53	±8.07	6.08	3.37	1,984.7	171.67
	$RDFS_s$	54.28	±382.46	54.04	±382.38	10.06	±13.42	6.12	2.61	4,557.19	2,789.54
	$Comb_s$	55.14	±382.35	54.88	±382.27	12.52	±17.02	6.37	3.46	5,043.68	3,171.05
entity-so	Core	16.32	±15	35.34	±51.01	17.49	±8.08	6.65	19.86	3,853.07	–
	Core⁻	16.14	±15.04	34.66	±49.83	12.28	±6.41	6.79	6.46	1,296.02	–
	seeAlso	16.17	±15.02	34.68	±49.82	13.01	±7.22	6.65	7.49	2,551.37	–
	sameAs	18.19	±17.4	56.14	±93.8	15.15	±11.48	6.68	8.53	1,776.64	393.93
	$RDFS_s$	19.71	±17.52	52.86	±73.08	14.19	±14.12	6.65	8.64	4,600.24	2,833.88
	$Comb_s$	21.78	±19.66	81.22	±122.66	16.84	±17.08	6.7	11.32	6,317.88	4,121

Table D.1: Detailed QWalk results for entity-* queries

	Setup	Term		Results		Time (s)		First (s)	HTTP	Data	Inferred
		avg.	σ	*avg.*	σ	*avg.*	σ				
star-0-3	Core	3.49	±4.46	2.64	±4.4	14.9	±16.5	7.43	13.76	2,099.61	–
	Core$^-$	3.49	±4.46	2.64	±4.4	7.47	±2.57	6.95	1.9	595.42	–
	seeAlso	3.49	±4.46	2.64	±4.4	7.94	±4.64	7.43	1.91	595.42	–
	sameAs	4.03	±6.4	8.76	±49.72	9.31	±6.82	7.44	2.31	3,098.49	54.58
	RDFS$_s$	3.49	±4.46	2.64	±4.4	10.88	±25.86	7.24	1.9	10,328.1	6,816.55
	Comb$_s$	4.03	±6.4	8.76	±49.72	12.83	±31.08	7.78	2.33	9,920.43	6,876.52
star-1-2	Core	9.05	±31.78	11.35	±53.6	65.84	±391.73	7.25	13.26	1,709.11	–
	Core$^-$	9.05	±31.78	11.35	±53.6	7.08	±2.18	6.72	1.82	48.74	–
	seeAlso	9.08	±31.79	11.68	±53.8	8.68	±10.44	6.65	2.21	62.81	–
	sameAs	9.53	±31.83	13.02	±54.21	8.15	±3.94	6.71	2.4	323.32	52.97
	RDFS$_s$	12.56	±53.7	644	±5,028.56	9.07	±12.49	6.88	1.82	856.34	556.48
	Comb$_s$	13.06	±53.7	645.95	±5,028.32	10.98	±17.91	6.68	2.74	1,354.61	1,017.95

Table D.2: Detailed QWalk results for star-* queries

	Setup	Term		Results		Time (s)		First (s)	HTTP	Data	Inferred
		avg.	σ	*avg.*	σ	*avg.*	σ				
s-path-2	Core	6.89	±38.25	6.35	±38.27	14.42	±34.39	7.05	14.02	575.45	–
	Core$^-$	6.89	±38.25	6.35	±38.27	8.86	±4.98	6.38	2.8	236.39	–
	seeAlso	6.89	±38.25	6.35	±38.27	9.38	±5.35	6.37	3.08	236.45	–
	sameAs	7.08	±38.24	6.74	±38.32	9.62	±5.74	6.25	3.52	757.2	71.17
	RDFS$_s$	7.79	±38.17	7.62	±38.28	9.14	±5.25	6.49	2.86	2,016.91	1,325.12
	Comb$_s$	8	±38.17	8.03	±38.32	10.25	±5.95	6.43	3.98	2,214.55	1,502.45
s-path-3	Core	48.14	±237.92	28.55	±122.05	16.21	±22.62	6.61	16.53	2,605.86	–
	Core$^-$	48.1	±237.89	28.43	±121.89	11.6	±6.73	6.89	4.59	1,302.96	–
	seeAlso	49.35	±237.82	29.06	±121.83	11.37	±8.02	6.51	5.94	1,305.53	–
	sameAs	48.14	±237.89	28.47	±121.88	11.59	±5.82	6.74	4.71	4,612.86	4.04
	RDFS$_s$	48.69	±237.78	29.02	±121.76	11.94	±6.84	7.1	4.61	13,502.16	8,601.94
	Comb$_s$	51.59	±238.09	30.49	±121.85	14.54	±17.97	6.88	7.76	13,329.9	8,657.35

Table D.3: Detailed QWalk results for *-path-* queries

	Setup	Term		Results		Time (s)		First (s)	HTTP	Data	Inferred
		avg.	σ	*avg.*	σ	*avg.*	σ				
star-2-1	CORE	5.53	±12.82	7.44	±30.09	12.77	±16.11	6.41	11.79	515.37	–
	CORE^{-}	5.51	±12.82	7.43	±30.09	6.63	±2.13	6.25	1.73	104.11	–
	SEEALSO	5.51	±12.82	7.43	±30.09	7.46	±3.37	6.23	1.93	104.5	–
	SAMEAS	5.66	±12.8	7.57	±30.07	7.25	±2.69	6.57	1.79	372.14	14.16
	RDFS$_S$	6.1	±13.5	17.04	±87.46	7.98	±10.98	6.31	1.73	814.74	409
	COMB$_S$	6.24	±13.47	17.19	±87.44	9.84	±13.66	6.99	1.99	797.06	435.4
star-3-0	CORE	2.2	±0.79	1.15	±0.56	10.18	±5.65	7.1	6.79	1,242.77	–
	CORE^{-}	2.2	±0.79	1.15	±0.56	7.16	±3.26	6.69	1.53	949.88	–
	SEEALSO	2.2	±0.79	1.15	±0.56	7.76	±3.99	6.75	1.76	964.27	–
	SAMEAS	2.29	±1.15	1.24	±1.01	8.33	±5.6	6.93	1.83	2,138.38	68.39
	RDFS$_S$	3.67	±2.57	2.7	±2.61	11.46	±26.25	7.51	1.53	8,817.83	6,742.48
	COMB$_S$	3.77	±2.66	2.8	±2.7	13.8	±36.07	9.58	1.95	9,411.11	7,203.71

Table D.4: Detailed QWalk results for star-* queries

	Setup	Term		Results		Time (s)		First (s)	HTTP	Data	Inferred
		avg.	σ	*avg.*	σ	*avg.*	σ				
o-path-2	Core	96.02	±379.55	94.26	±376.01	68.52	±262.1	7.34	107.06	2,942.29	–
	Core^{-}	96.02	±379.55	94.26	±376.01	11.23	±7.84	7.16	4.18	1,260.85	–
	seeAlso	96.02	±379.55	94.26	±376.01	12.15	±7.95	6.87	4.52	1,261.71	–
	sameAs	140.79	±566.69	139.19	±564.44	13.19	±9.95	7.24	6.21	7,333.55	1,146.18
	RDFS$_s$	96.08	±379.59	96.03	±377.18	15.09	±23.65	7.14	4.18	14,604.69	10,073.15
	Comb$_s$	140.87	±566.71	140.98	±565.07	20.47	±36.77	7.32	6.69	18,734.92	12,521.85
o-path-3	Core	83.49	±268.83	86.06	±288.12	90.78	±268.35	7.41	157.46	7,293.86	–
	Core^{-}	83.51	±268.87	86.09	±288.15	15.1	±8.86	7.9	7.43	1,105.51	–
	seeAlso	83.51	±268.87	86.09	±288.15	14.75	±8.35	7.51	7.49	1,105.66	–
	sameAs	83.51	±268.87	86.09	±288.15	14.16	±8.98	7.09	7.43	3,854.49	–
	RDFS$_s$	83.51	±268.87	86.09	±288.15	17.1	±15.34	7.14	7.43	9,336.74	5,481.91
	Comb$_s$	83.51	±268.87	86.09	±288.15	18.38	±18.07	8.01	7.51	9,332.71	5,477.31

Table D.5: Detailed QWalk results for star-* queries

LIST OF FIGURES

LIST OF TABLES

LIST OF QUERIES

ACRONYMS

FOAF Friend Of A Friend

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

LTBQE linked-traversal based query execution

LOD Linked Open Data

MIME Multipurpose Internet Mail Extensions

OWL Web Ontology Language

RDF Resource Description Framework

RDFS RDF Vocabulary Description Language: RDF Schema

Turtle Terse RDF Triple Language

URI Uniform Resource Identifier

WWW World Wide Web