Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
von der Fakultät für Wirtschaftswissenschaften
des Karlsruher Instituts für Technologie (KIT)
genehmigte Dissertation von
Steffen Walter Stadtmüller, M.Sc.

DYNAMIC INTERACTION AND MANIPULATION OF WEB RESOURCES

STEFFEN WALTER STADTMÜLLER

Tag der mündlichen Prüfung: 01.12.2015
Referent: Prof. Dr. Rudi Studer
Korreferent: Prof. Dr. Jorge Cardoso

Karlsruhe, 2015

This document was created on January 26, 2016.

ABSTRACT

The World Wide Web is the single largest information system of mankind, which allows for the ubiquitous access to data and services for over 3 Billion people. Apart from human readable Web pages providers offer APIs, which allows applications to combine and include available information and functionality to fulfill arbitrary tasks.

However, there is little coordination between providers, and applications have to handle heterogeneous interfaces with unaligned vocabularies. Linked Open data unifies an interaction model for the consumption of information of graph-structured interlinked data resources and schemata. Thus, applications can be designed to follow links to discover relevant information and align different resources using reasoning features leveraging the schemata. Due to the unpredictable and dynamic nature of the Web, applications have to interpret discovered schema information at runtime and evaluate queries directly over dereferenced resources without relying on pre-existing index structures over the data. At the same time applications must exhibit high-performance characteristics with regard to data processing and retrieval to achieve short response times and a fluent interaction with users.

In this thesis we describe how we join methods for evaluating queries over interlinked resources via link traversal with approaches for the integration of data over interlinked schemata via reasoning. In particular, we show how declarative rule-based programs can be used to specify desired dynamic interactions with Web resources. We introduce a system with a parallel push-based execution model that allows for the balancing of the heterogeneous workload of resource retrieval and data processing. Specifically, we show different alternatives to implement such an execution model to avoid communication overhead in the face of a parallel execution and detail how the retrieval of remote information can be integrated without hampering the data processing. Our approach allows for the on-the-fly alignment and processing of dynamically retrieved data in a streaming fashion including incremental query answering. We conduct experiments to analyse the behaviour of a fully implemented system that realises the proposed interaction model with respect to different workloads for data processing and resource retrieval.

As further contribution we go beyond the simple consumption of exposed information by enabling rules to define intended manipulations of remote resources. We describe the synergies of the combination of Linked Data principles as data representation model and Representational State Transfer as interaction model. In particular, we formalise the combination of both technologies as state transition systems. Build upon such state transition systems we describe how rule programs can be employed to effectively define the manipulation of remote Web resources for integration of functionality from various providers. Specifically, the effected interactions can be derived from at runtime identified information to enable applications to dynamically follow links. We show how our ap-

proach can be applied to achieve a dynamically reacting system for Web-based applications, thus accommodating the constantly changing environment of the Web. Further we conduct experiments to provide evidence that the consideration of resource manipulations beyond data consumption does not impede the performance of our system.

Web-based applications can be built to discover resources at runtime via link traversal. However, an entry point for applications to start the interaction with web resource must be identified. We detail how graph patterns can be employed to describe resources, thus enabling to search for such entry points. Specifically, resources can be identified that provide the required information for an application to begin the interaction with a Web API. Further, we detail methods to realise a system for resource search build upon graph patterns in a scalable distributed manner, which we experimentally confirm.

Overall the contributions of this thesis focus on the development of applications that leverage remote Web resources dynamically. Given the increasing importance of Web-based applications, advances in this field can provide valuable insights for the use of remote knowledge on the Web.

PUBLICATIONS

Text as well as figures in this thesis have partly been already published. That is, the thesis is based on the following papers:

Steffen Stadtmüller and Barry Norton. Scalable discovery of linked APIs. *International Journal of Metadata and Semantics and Ontologies*, 8(2):95–105, 2013. [106]

Steffen Stadtmüller, Sebastian Speiser, Andreas Harth, and Rudi Studer. Data-Fu: A language and an interpreter for interaction with read/write linked data. In *Proceedings of the International Conference on World Wide Web*, Rio de Janeiro, Brazil, 2013. ACM International Conference Proceeding Series. [108]

Steffen Stadtmüller, Jorge Cardoso, and Martin Junghans. Service semantics. In Jorge Cardoso, Hansjörg Fromm, Stefan Nickel, Gerhard Satzger, Rudi Studer,and Christof Weinhardt, editors, *Fundamentals of Service Systems*. Springer, 2015. [110]

ACKNOWLEDGMENTS

Without the advice, support and encouragement I received from so many people, this dissertation would not have been possible. First and foremost I want to thank my advisor Prof. Dr. Rudi Studer for providing me with the opportunity and granting the freedom to do this research. His guidance was invaluable and always a great source of inspiration. It has been an great honor to be a member of his research group at the institute AIFB. I would also like to thank Dr. Andreas Harth, my frequent co-author and patient advisor that supported and motivated me during my work on this thesis.

Many thanks also to Prof. Dr. Jorge Cardoso and Prof. Dr. Gerhard Satzger for their interest and their consent to serve as members of the board of examiners. The many fruitful discussions with them provided me with valuable insights for my work.

The work with all my friends and colleagues at the Knowledge Management research group at AIFB and the institute KSRI has been very inspirational and helped to improve my work. I am truly grateful that I had the chance to work with so many talented individuals in an incredibly friendly and supportive atmosphere.

Especially, I would like to extend my deepest gratitude to my parents, Renate and Walter Stadtmüller and my whole family, who always encouraged and supported me in my endeavors. Without them I would not be where I am today.

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation . . . 1
  - 1.1.1 Challenges . . . 2
  - 1.1.2 Scope of the Thesis . . . 4
  - 1.1.3 Dynamic Web Resources . . . 5
- 1.2 Scenario . . . 6
- 1.3 Hypotheses . . . 7
- 1.4 Contributions . . . 9
- 1.5 Outline . . . 10

2 FOUNDATIONS 12
- 2.1 Overview . . . 12
- 2.2 Linked Data . . . 12
  - 2.2.1 Resource Description Framework . . . 14
  - 2.2.2 Interlinked Resources . . . 18
  - 2.2.3 Basic Graph Pattern . . . 21
- 2.3 Representational State Transfer . . . 25
  - 2.3.1 Operation-oriented API . . . 27
  - 2.3.2 Resource-oriented API . . . 28

3 PARALLEL PROCESSING OF WEB RESOURCES 34
- 3.1 Introduction . . . 34
  - 3.1.1 Challenges . . . 36
  - 3.1.2 Contributions . . . 37
- 3.2 Rule-based Programs . . . 38
- 3.3 Processing Architecture . . . 45
  - 3.3.1 Network Request Component . . . 48
  - 3.3.2 Physical Operator Plan Component . . . 49
  - 3.3.3 EquiJoin Operator . . . 53
- 3.4 Coordination of Data Processing and Network Requests . . . 57
  - 3.4.1 Threading Models . . . 58
  - 3.4.2 Blocking Worker Control . . . 60
  - 3.4.3 Adaptive Processing . . . 65
  - 3.4.4 Handling Duplicates . . . 67
- 3.5 Experiments . . . 71
  - 3.5.1 CPU-bound Tasks . . . 74
  - 3.5.2 I/O-bound Tasks . . . 82
  - 3.5.3 Mixed CPU- and I/O-bound Tasks . . . 85
- 3.6 Related Work . . . 88
- 3.7 Summary and Future Work . . . 90

4 DYNAMIC MANIPULATION OF WEB RESOURCES 93
4.1 Introduction . . . 93
4.1.1 Challenges . . . 96
4.1.2 Contributions . . . 97
4.2 Combining Linked Data and REST . . . 98
4.2.1 URI-identified Resources . . . 99
4.2.2 Interaction Methods . . . 99
4.2.3 Hypermedia Links . . . 102
4.2.4 Wrapping APIs . . . 104
4.3 Linked API Interaction Model . . . 107
4.4 The Linked Data-Fu Language . . . 110
4.4.1 Program Execution . . . 118
4.4.2 Non-Deterministic Behaviour . . . 120
4.4.3 Repeated Program Execution . . . 125
4.5 Experiments . . . 129
4.5.1 Number Retrieval and Manipulation . . . 130
4.5.2 Game of Life . . . 137
4.6 Related Work . . . 142
4.7 Summary and Future Work . . . 143

5 WEB RESOURCE SEARCH 146
5.1 Introduction . . . 146
5.1.1 Challenges . . . 148
5.1.2 Contributions . . . 149
5.2 Graph Pattern Descriptions . . . 149
5.3 Matching . . . 153
5.4 Ranking . . . 154
5.4.1 Containment-based metric . . . 155
5.4.2 Vocabulary-based metric . . . 158
5.5 Search Architecture . . . 161
5.6 Experiments . . . 164
5.6.1 Distributed Search . . . 165
5.6.2 Containment Ratio Calculation . . . 168
5.7 Related Work . . . 170
5.8 Summary and Future Work . . . 171

6 CONCLUSION 173
6.1 Summary . . . 173
6.2 Future Work . . . 176

BIBLIOGRAPHY 178

List of Figures 189

List of Tables 191

List of Algorithms 193

Acronyms 194

A APPENDIX: ADDENDUM 196
A.1 OWL LD Operator Plan 196
A.2 Experiment Result Details 198
A.3 Rule Program Examples 216

1

INTRODUCTION

1.1 MOTIVATION

The World Wide Web (Web) has become the single largest information system of mankind, providing ubiquitous access to data and services for over 3 Billion people (42.3% of the world population)[1]. With over 50 billion indexed Web pages[2] a strong focus of the Web remains the provisioning of interlinked human-readable documents.

Going beyond the traditional human-readable web, Tim Berners-Lee, the inventor of the Web, described 2001 his vision of the transformation of the Web to the *Semantic Web* [12], which allows automated agents to fulfill sophisticated tasks for humans by accessing, combining and interacting with data on the Web. Such agents would rely on the possibility to process the semantics of the data and a defined structure of the meaningful part of the Web. Consequently Linked Data (LD) has become a popular way to publish data. LD is identified with a set of principles and best practices regarding the use of certain technologies for the publication of data. The core idea of LD is that links connect entities (rather than Web pages). Links are typed to allow a characterization of the relationship of the entities. Schemata that are also available on the Web provide reusable vocabularies for the publishing of information as LD. In combination with the push to opening up public sector data as well as data from other domains LD has seen tremendous growth in recent years [13] with over thousand publicly available data sets [96], which contain at least 4 Billion statements, as confirmed by recent crawls[3].

However, general purpose agents that are able to perform arbitrary tasks with such datasets remain elusive. There is a trend to provide Application Programming Interfaces (API) and services on the Web that allow the development of client applications performing a specific task. Amazon AWS Marketplace[4], which was launched in 2012, has up to now more than 1 250 services available. Noor et al. [78] found that almost 6 000 services are already available on the Web. The study carried out consisted of searching the Web using a customized service crawler engine to find websites that offered services. Additionally, ProgrammableWeb[5], a popular registry for Web-based APIs lists over 13 000 inter-

[1] http://www.internetworldstats.com/, retrieved 2015-03-15.
[2] http://www.worldwidewebsize.com/; retrieved 2015-03-15.
[3] http://km.aifb.kit.edu/projects/btc-2014/; retrieved 2015-03-15
[4] http://aws.amazon.com/marketplace; retrieved 2015-03-15
[5] http://programmableweb.com; retrieved 2015-03-15

faces. Statistics of ProgrammableWeb show an exponential growth of the service ecosystem on the Web that can be seen over the last years. In principle, the possibility to combine the abundantly available data sources and functionalities of Web services open up a broad spectrum of opportunities to develop complex specialised client applications.

Independently from the question if the trend for APIs and services represents the first step towards autonomous general purpose agents or has to be considered as a move away from the original vision, the challenges that arise in the development of specialised client applications are ultimately also hurdles for the effort to create autonomous agents.

1.1.1 *Challenges*

Runtime Requirements. The general advice with regard to response times of applications, i.e., the time a user of an application has to wait after a command to see the effect of the command, is summarized as follows [74, 77]:

- *0.1 second:* The limit for users to feel that they directly manipulate objects in an User Interface (UI) of an instantaneously reacting system.
- *1 second:* The limit for users to feel that they are freely navigating, without interruption of the users flow of thought because of the response time. Delay between 0.2 and 1 second results in the user noticing a delay due to the system working on the command, rather then directly manipulating the system.
- *10 seconds:* The limit for users to keep their attention on the currently performed task. A delay above 10 seconds implies that users might have to reorient themselves after returning to the UI.

Although these limits can only be seen as general guidelines, for which mitigating approaches like wait time indicators exist, their application on Web-based applications, where response time is additionally affected by network latency and bandwidth, highlights the importance of the consideration of processing time. Especially the response time of an application that leverages and combines a multitude of data sources and Web services should not simply be the cumulative sum of the required runtime to interact with underlying sources and APIs.

Purely network related tasks like the retrieval of data from Web sources and the invocation of a Web service influence runtime by themselves for the most part by the delay caused by network latency and bandwidth limitations. However, data retrieval and service invocation require less computational resources from a client application than processing related tasks e.g., query answering over the retrieved data, inferencing, data integration and decision making with regard to the steps to be performed to achieve the objective of the application. Consequently the parallel execution of network and processing tasks is necessary to efficiently leverage the available computation resources on the system on which a Web-based application is executed.

The management of several in parallel executed tasks can, however, also cause additional system overhead, especially since the different network tasks and pro-

cessing tasks can be interdependent. E.g., a query over a data set can locally only be completed after the data set was retrieved. The design of a parallel system for Web-based applications has to take into account the runtime requirements of the application, which is a hard but important challenge.

Entropy. Understood as a measure of unpredictability [98] the entropy[6] entailed in the use of Web-based data and services represents an important challenge in the development of client applications. An autonomous agent that generally acts on the Web, would have to have a plethora of information available to be able to decide which of the available datasets and services are most helpful for achieving the intended goal of the agent, and to be able to retrieve the data, invoke the services and finally to interpret the received messages and data for further decisions.

The required information can pertain to the semantics of the functionality of a service. Information about what can be achieved by a service are especially relevant when an agent has to identify which service to use for a specific task. Furthermore there can be multiple non-functional properties, that can be relevant for the decision which available services should be used, e.g., pricing, availability or legal constraints.

Another aspect for which information is required is the actual invocation and use of functionality. There are multiple interaction schemata, API designs, data formats and annotations possible, the variety of which drives the entropy an autonomous agent would have to handle.

Even if we consider a manually crafted application for a specific task, instead of a general purpose agent, the entropy remains an important challenge, as the possible heterogeneity of the involved datasets and services translates into a large amount of information that a developer needs to gather and integrate when developing the application.

To mitigate the variety one can rely on convention and descriptions. Convention implies that some aspects of the Web-based functionality potentially follow an agreed upon standard, on which an agent or developer can rely. Although not all available datasets or services might follow such an agreement, it allows to restrict a client application to a subset of possibilities, thus reducing the entropy. A prime example for such an agreement is the use of Representational State Transfer (REST), which has become the predominantly used interaction paradigm on the Web (see Section 2.3). REST constrains the available operations of an API, thus limiting the variety of possibilities how to us a service.

Descriptions rely on the idea that necessary information to use a given functionality is directly provided for client applications. Such descriptions can be human-readable documents (e.g., Web pages) that a developer can use to gain an understanding of the functionality and how to use it. In contrast machine-readable descriptions are based on a formal specification and aim at an au-

[6] Formally entropy is defined in [98] as the expected value of the information contained in a message as $H = -\sum_{i=0}^{N-1} p_i log_2 p_i$ with p_i the probability of a given symbol. Thus, entropy provides a way to estimate the average minimum number of bits necessary to encode a string of symbols. Here we use the term to describe the average information necessary to conduct an interaction with Web-based functionality and data.

tomatic consumption and interpretation of a client application. While human-readable descriptions can rely on the human for interpretation, a machine readable description only works for client applications that are build with the corresponding specification in mind. Thus, a client application might not be able to interpret a description, which renders the description useless for this application. The amount of existing approaches (see e.g., [93, 129]) consequently just encodes the existing variety and does not combat entropy, as an application has either be able to deal with the very large number of description approaches or be limited to a smaller subset. Ultimately the lacking of a shared understanding (i.e., convention) on what description approach to use, prevents the reduction of the entropy.

LD leverages schemata and ontologies to provide "a formal, explicit specification of a shared conceptualization" [113] expressing the meaning of data. Thus client applications can automatically infer the relevance of data for their intended task. As schemata are interlinked with mappings between concepts (see Section 2.2), a client application can to some extent also leverage previously unknown schemata. Therefore LD provides mechanisms to cope with entropy with regard to the interpretation of the data.

The use and combination of approaches to handle the entropy in a Web environment is an important challenge that has to be addressed to ease the development of Web-based applications.

1.1.2 *Scope of the Thesis*

In this thesis we design and study methods to interact with and process data and services in a distributed Web-environment that minimise application response time. Specifically we address methods that tackle the entropy of dynamic Web resources (see Section 1.1.3) with a performance that allows for high fidelity applications. We focus on REST and LD as well-adopted core Web-technologies that allow to build applications in the face of the entropy of a Web-ecosystem with services and data. Specifically we focus on the use of rule-based programs for the interaction and manipulation of Web resources and the use of graph patterns to support the search for relevant resources[7].

Our approach does not target the development of autonomous Artificial Intelligence (AI) agents, but centers around methods for the development of client applications, that could also be leveraged by such agents. Furthermore, we do not aim to provide an additional approach for Web Service descriptions, but focus on commonalities of existing description approaches in the context of REST and LOD.

Also out of scope of the thesis are issues related with access control and trust when interacting with Web resources. We assume that wherever necessary, mechanisms are in place to ensure authorised access, e.g., OAuth2.0[8], and the trustworthiness of sources, e.g., a web of trust [28].

[7] We introduce rules and graph patterns in the corresponding Chapters 3-5.

[8] http://oauth.net/2/; retrieved 2015-04-10

1.1.3 *Dynamic Web Resources*

Both LD and REST are focused on resources[9]:

- LOD centers around the publication and consumption of data about interlinked resources.
- REST establishes a service architecture, where the invocation and use of a service is synonymous with interacting and manipulating resources.

In the context of this thesis we define *Web resources* based on the Web characterization terminology of the World Wide Web Consortium (W3C)[10]:

➳ Definition 1: Web Resource

A Web resource is any physical or abstract entity that has identity. A Web resource is identified by a Uniform Resource Identifier (URI), which allows the entity to be accessed by any implemented version of the Hypertext Transfer Protocol (HTTP) as part of the protocol stack, either directly or as data about the entity via an intermediary. Specifically, an entity can be

- a real world object (e.g., concert ticket, sports team)
- a digital object (e.g., blog post, Web page)
- a concept (e.g., membership, payment)
- a collection of other resources (e.g., list of all teams in sports league)

Resources and specifically their properties can change over time. The W3C specifically defines[11] *resource manifestations* as "a rendition of a resource at a specific point in time and space", where a conceptual mapping exists between the resource and its manifestation.

The change of a resource can be slow, e.g., resulting from curation of the data, or fast paced, because the properties of the resource are inherently fast changing.

✎ Example 1

The data describing a sports stadium (which is available as Web resource) might be updated after renovations with the new amount of seats in the stadium. However, if the resource is updated with data from sensors in the stadium providing information about the number of people currently in the stadium, the resource might change several times per second.

The dynamicity of Web resources is the reason for building Web-based applications: If a dataset is completely static and never changes, there is no reason to use the data remotely over the Web. Instead one could simply download the data once and use them locally, thus avoiding network delays. It is the inherent constant change of the Web resources that makes them valuable, as the changes allow to leverage up-to-date information from a variety of sources.

[9] A detailed introduction in both technologies, REST and LD, is provided in Chapter 2.
[10] http://www.w3.org/1999/05/WCA-terms/; retrieved 2015-04-10.
[11] http://www.w3.org/1999/05/WCA-terms/; retrieved 2015-04-10.

The constant change of Web resources, however, also amplifies the challenges to use them for applications. The entropy for a client application retrieving a resource stays high even after the first retrieval, as the client might encounter different information every time. Not only the properties of a resource might change, but also links to other resources and schemata. Therefore, clients require a dynamic interaction and manipulation approach that allows to decide at runtime which links to follow and how to interpret the available schemata.

The entropy caused by the change in the resources also increases the challenge with respect to the runtime requirements of an application: As resources are retrieved dynamically a parallel processing model must be able to process data at arrival and cannot rely on preexisting indexes over the data. Also the schema information to interpret the data has to be processed on-the-fly, as no knowledge about the schemata can be assumed.

1.2 SCENARIO

Throughout the thesis we use examples from the fictitious company ACME. The business model of ACME is derived from existing companies like eventim[12] and last.fm[13] that act as event recommendation and ticket agent.

> ✎ **Example 2**
>
> ACME is an online ticket agency that sells tickets to various events like concerts and sports matches. Customers and users can register with ACME and create a profile with information about their interests, like their favorite music genre or sports team. As additional service to foster customer retention, ACME provides the users up-to-date information relevant to the provided interests, like scores of matches of their favorite sport team or upcoming concerts of their favorite band. To provide such information ACME relies on third party Web sources, from which ACME gathers relevant data.
>
> Additionally, users can also provide information in their profiles about other social networks and message boards on which the users have accounts. ACME uses the account information to disseminate offers about special sales over multiple channels.
>
> The services of ACME can be used via a web site, but ACME also offers an API to allow third party applications to make use of the provided functionality and consequently increase outreach and available customer basis. ACME functions as service and data consumer as well as provider on the Web and relies on dynamic approaches for the interaction and manipulation of Web resources, to allow for a flexible change of the used information sources as well as dynamic dissemination of information.

We assume the URI namespaces and prefixes of ACME as listed in Table 1

[12] http://www.eventim.com/; retrieved 2015-04-10
[13] http://www.last.fm/; retrieved 2015-04-10

Table 1: URI namespaces and prefixes of the fictitious company ACME used throughout the thesis

Prefix	URI namespace
`acme:`	`http://acme.example.org/api/`
`data:`	`http://acme.example.org/data/`
`p:`	`http://acme.example.org/vocabulary#`

1.3 HYPOTHESES

To address the challenges outlined in Section 1.1 we define the following hypotheses with associated research questions:

> □ **Hypothesis 1**
>
> Declarative rule-based programs can be utilised to define desired dynamic retrieval and processing of Web resources for client applications in such a way that retrieval and processing can be executed in a highly parallel streaming fashion, thus enabling applications with short runtimes. (Chapter 3)

Associated with Hypothesis 1 are the following research questions:

1. How can dynamic interactions with Web resources based on on-the-fly discovered schemata be specified in a rule-based language?
2. How can a data-driven parallel execution model for the interactions be designed that is capable of on-the-fly processing of arriving data and schema information?
3. How do different implementations of data-driven execution models perform with respect to processing time in the face of intertwined data processing and network lookups?

> □ **Hypothesis 2**
>
> The principles of Linked Open Data and Representational State Transfer can be combined to an interaction model for APIs based on state transitions that allows to design rule-based programs also for the manipulation of Web resources, which are dynamically discovered via link traversal, while preserving short runtimes. (Chapter 4)

Associated with Hypothesis 2 are the following research questions:

1. Can the interaction models of LD and REST be combined and formalised as a state transition system?
2. How can manipulations of dynamically via link traversal discovered Web resources be defined based on a state transition interaction model?
3. What are the execution semantics of a declarative rule-based program for the manipulation of Web resources?

□ **Hypothesis 3**

Graph patterns to describe the possible interactions and manipulations of Web resources can be leveraged to allow for the scalable search of resources, which can serve as entry point for an interaction. (Chapter 5)

Associated with Hypothesis 3 are the following research questions:

1. How can potential interactions and manipulations of Web resources be described with graph patterns with respect to applied operations?
2. How can graph pattern-based descriptions be used to search for resources with desired functionality?
3. How can search results be identified in a scalable fashion, given a graph pattern-based search request?

Hypothesis 1 is mainly concerned with the realisation of a processing model for rule-based programs that includes query answering, network resource retrieval and inferencing to achieve Web-based applications with low response times. However, the approach build upon Hypothesis 1 as described in Chapter 3 respects the requirements resulting from a dynamically changing Web environment. Therefore, the specific focus is on the on-the-fly processing of arriving data in a streaming fashion, which includes the dynamic inclusion of schema information discovered at runtime.

Hypothesis 2 goes beyond the simple retrieval of resources and targets also the manipulation, i.e., effected change, of resources. The corresponding approach in Chapter 4 tackles the entropy-related challenges by combining LOD and REST, while specifically retaining the advantages with respect to processing time of applications, as developed for Hypothesis 1. The approach further addresses entropy with methods to describe desired dynamic link traversal; i.e., applications are enabled to follow links that are unknown at design-time to achieve desired processing goals.

Hypothesis 3 addresses the application of description methods to mitigate existing entropy for the search of resources. While Hypotheses 1 and 2 do not presuppose the existence of descriptions for processing data and services in client applications, Hypothesis 3 specifically focuses on the high-level task of resource search build upon the existence of descriptions. However, also the runtime requirements of the approach to search, based on Hypothesis 3, and specifically the scalability of the approach with respect to the amount of available services remains in focus in Chapter 5.

1.4 CONTRIBUTIONS

With regard to the aforementioned hypotheses, this thesis provides the following contributions:

> ☞ **Contribution for Hypothesis 1**
>
> Parallel data-driven push-based execution model for on-the-fly processing of rule-based programs to access Web resources.

We describe an execution model for the parallel evaluation of rule-based programs that allow inferencing over distributed Web resources, as well as query answering over the processed data. We describe a rule language that intertwines data processing and network lookups, which can be evaluated with the execution model. Our approach realises the combination of data-driven operator scheduling with a push model. The operator scheduling avoids overhead of inter-process communication. The push model allows data to be processed as soon as it arrives, thus specifically caters to applications build upon network lookups in a Web environment. In particular, we detail different implementations of the proposed execution model and describe how the workload resulting from data processing and data retrieval can be balanced.

> ☞ **Contribution for Hypothesis 2**
>
> Formal model of Linked Data-based REST APIs for dynamic manipulation of Web resources.

Based on our work in [108] we introduce a formal model for APIs based on the principles of REST and LD as state transition system, which is the basis for the extension of the introduced rule-based programs for the manipulation of Web resources. Specifically, the formal model allows to specify intended effects of a program to manipulate resources as a reaction to resource manifestations at runtime, thus addressing the required dynamicity for the interaction and manipulation of Web resources. Specifically we provide the execution semantic of a rule program to manipulate Web resources and describe approaches to mitigate potential non-deterministic behaviour. Furthermore, our approach enables to specify at design time the manipulation of resources that are discovered at runtime via link traversal. The extension of the programs for the manipulation of Web resources preserves the applicability of the parallel execution model (Contribution 1).

> ☞ **Contribution for Hypothesis 3**
>
> Methods to realise a scalable resource search based on graph pattern descriptions of Web resources.

Based on our work in [106, 110] we describe how an often found commonality of approaches to describe resource-oriented services (i.e., the application of graph patterns) can be used to effectively encode the interactions and manipulations

that are possible for a given LD Web resource. Further we show how such descriptions can be exploited for a scalable search system of resources with respect to the amount of available resources. For the search of resources we consider the specification of search requests, the matching of search requests against resource descriptions, as well as ranking of matching results with respect to the search request.

Our research for the described contributions has led to the development of the end-to-end data processing system Linked Data-Fu (LD-Fu)[14], which consists of a Notation3[15]-based language to specify programs with logical and production rules as well as an engine to evaluate such programs. LD-Fu supports query answering, reasoning and HTTP-based data access and manipulation. Amongst other deployments LD-Fu is used in various research projects, e.g., I-VISION[16] and ARVIDA[17], where it manages the communication and integration of heterogeneous data sources in virtual reality environments.

Experiments conducted with LD-Fu throughout this thesis are based on version 0.9.3 of the engine.

1.5 OUTLINE

The remainder of this thesis consists of five chapters, which convey Foundations and the Hypotheses (1) - (3).

❷ *Chapter 2 – Foundations*
In Chapter 2 we provide preliminaries for our approaches in Chapters 3- 5. In particular, we introduce the principles and technologies associated with LD as well as REST-based architectures.

❸ *Chapter 3 – Parallel Processing of Web Resources*
In Chapter 3 we present a parallel data-driven processing model for rule-based programs, that combines inferencing capabilities with the network retrieval of Web resources. We provide alternatives for the implementation of such a data-driven processing model and compare their behaviour with respect to the involvement of Web resource retrieval.

❹ *Chapter 4 – Dynamic Manipulation of Web Resources*
In Chapter 4 we go beyond the retrieval and processing of resources with programs that effect changes in Web resources. For this purpose we introduce a formal model for APIs based on the principles of REST and LD. We show how programs can be defined that manipulate Web resources dynamically as subject to conditions regarding other resources and provide the execution semantics of such programs.

[14] http://linked-data-fu.github.io/; retrieved 2015-04-10
[15] http://www.w3.org/TeamSubmission/n3/; retrieved 2015-04-10
[16] http://www.ivision-project.eu/; retrieved 2015-04-10
[17] http://www.arvida.de/; retrieved 2015-04-10

❺ *Chapter 5 – Web Resource Search*
In Chapter 5 we describe how graph pattern can be used to encode possible interactions and manipulations of Web resources. We show how graph pattern-based descriptions can be leveraged in resource search systems. Specifically we introduce algorithms on-top of the graph patterns for matching and ranking of resources given a search request.

❻ *Chapter 6 – Conclusion*
Last, we summarize our contributions and results in Chapter 6. Further, we give an outlook on important future work.

2

FOUNDATIONS

2.1 OVERVIEW

Now we present the preliminaries for the remainder of the thesis. In particular, we introduce LD and its associated core technologies in Section 2.2. The technologies described in the context of LD provide the data model for Web resources, as well as basic graph pattern as foundation for queries, rules and descriptions.

Furthermore, we introduce REST in Section 2.3 in the context of Web-based APIs and services. Differentiated from operation-oriented approaches, REST provides the general resource-oriented model for the interaction with Web resources.

Where necessary, we expand on specific aspects of the technologies introduced in this Chapter throughout the thesis.

2.2 LINKED DATA

With the goal to develop ways to allow computers to interpret (sometimes termed understand) information on the Web, the W3C introduced the concept of a Semantic Web. The objective behind the introduction of the Semantic Web is to provide solutions for data integration and interoperability. The Semantic Web identifies a set of technologies and standards that form the basic building blocks of an infrastructure that supports the vision of Web-ecosystem for applications.

Linked Data is a subset of the Semantic Web that focuses on the publication and consumption of interlinked Web resources in a Semantic Web architecture. Such an architecture implies the commitment to the use of URIs to denote things as well as a set standards for data provision and queries. In particular, Linked Data adheres to the following four design principles[18] [13]:

- Use URIs as names for "things".
- Use HTTP URIs so that people can lookup the names.
- When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)
- Include links to other URIs in the descriptions to allow people to discover more "things".

[18] http://www.w3.org/DesignIssues/LinkedData.html; retreived 2015-04-10

The amount of published Linked Data on the Web has been increasing rapidly in resent years. The domains covered by these publicly available datasets include life sciences (e.g., PubMed[19] and Drugbank[20]), multimedia and entertainment (e.g., LastFM[21] and BBC Music[22]), publications (e.g., O'Reilly Products[23] and DBLP[24]), and geospatial services (e.g., Geo Names[25] and GADM[26]). There are also cross-domain datasets that provide encyclopedic information about a variety of topics like DBpedia[27] and Wikidata[28]. The available data is not only offered from the private sector, but also derived from user-generated content (e.g., Flickr). Also the public sector drives the push towards the provision of openly available structured information. E.g., the governments of the United States[29] and the United Kingdom[30] provide data about CO2 emissions, postal codes, mortality rates and crime statistics.

While the term Linked Data is often used synonymous with Linked Open Data, the latter specifically stresses the requirement that the resources are openly available. The openness of resources allows arbitrary applications and agents to be build upon the exposed LOD information, which is in agreement with the vision of the Semantic Web. Open data also fosters an interlinked structure of resources on the Web, as it allows a resource provider to find related resources of other providers and create links to these related resources. However, in the context of this thesis not all considered Web resources have to be openly available for everyone, which is often not even sensible.

> ✎ **Example 3**
>
> A resource provided by ACME, which contains information about the band Metallica can be open and freely available for everyone. In contrast a resource describing the invoice for the purchase by a specific user of a ticket to a concert with Metallica, should only be visible for even that user and system administrators of ACME.

Adhering to the Linked Data principles has many advantages that apply in the context of structured representation of data on the Web but also in the context of the formal descriptions of services, e.g., for service search, selection, composition, and analysis.

Advantages of Linked Data include:

- Explicit and simple data representation based on a common data representation (RDF) hides from underlying technologies and systems.

[19] http://pubmed.bio2rdf.org/; retrieved 2015-04-10.
[20] http://wifo5-03.informatik.uni-mannheim.de/drugbank/; retrieved 2015-04-10.
[21] http://lastfm.rdfize.com/; retrieved 2015-04-10.
[22] http://www.bbc.co.uk/music/; retrieved 2015-04-10.
[23] http://labs.oreilly.com/opmi.html; retrieved 2015-04-10.
[24] http://dblp.l3s.de/d2r/; retrieved 2015-04-10.
[25] http://www.geonames.org/; retrieved 2015-04-10.
[26] http://gadm.geovocab.org/; retrieved 2015-04-10.
[27] http://wiki.dbpedia.org/; retrieved 2015-04-10.
[28] http://www.wikidata.org/; retrieved 2015-04-10.
[29] http://www.data.gov/; retrieved 2015-04-10.
[30] http://www.data.gov.uk/; retrieved 2015-04-10.

- Simple publishing and consumption of Linked Data supported by easy-to-use systems and technologies.
- Cross-referencing allows for linking and referencing of existing data, via reuse of identifiers.
- Incremental data integration, i.e., add mappings, without extensive upfront effort.
- Decentralized distributed ownership and control facilitates adoption and scalability.
- Large and scalable systems can be developed more easily due to a loose coupling that Linked Data introduces with a common language and communication layer.

2.2.1 *Resource Description Framework*

Maintained by the W3C [69], the Resource Description Format (RDF) provides a common data model enabling the encoding of information that can be read and interpreted by computer applications. RDF provides a graph model for describing resources on the Web. The RDF model is based upon the idea of making statements about resources in the form of a subject-predicate-object expression, a triple in RDF terminology. Each element has the following meaning:

SUBJECT is the resource; the "thing" that is being described.

PREDICATE is an aspect about a resource that expresses the relationship between the subject and the object.

OBJECT is the value that is assigned to the predicate.

Triples are composed of URIs, blank nodes and literals as follows:

➳ Definition 2: RDF Term, RDF Triple, RDF Graph

Let $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}$ be pairwise disjoint infinite sets representing respectively URIs, blank nodes and literals.
The set of all RDF terms is denoted by $\mathcal{T} = \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$.
An RDF triple is defined as $\langle s, p, o \rangle \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \times \mathcal{B} \times \mathcal{L})$.
An RDF Graph is a finite set of RDF triples $G \subset (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \times \mathcal{B} \times \mathcal{L})$.
We denote $\mathcal{T}_G$ as the finite set of RDF terms in graph G.
For a set of RDF Graphs Γ, we denote $\dot{\bigcup}_{G \in \Gamma} G$ as the merged RDF graph, i.e., the union of all triples $\langle s, p, o \rangle \in G$, $\forall G \in \Gamma$ after forcing any shared blank nodes, which occur in more than one graph to be distinct[a].

[a] For the detailed definition of RDF merge see http://www.w3.org/TR/rdf11-mt/, retrieved 2015-04-10.

URIs are used to identify resources. Literals express values, such as strings, numbers, and dates. Literals can be typed with a data type, e.g., using the existing types from the XML Schema specification [15]. Untyped literals are interpreted as strings. Blank nodes serve as locally scoped identifiers for entities that do not have a URI.

RDF is a directed labeled graph data model: Subjects and objects are nodes, which are connected by a directed edge. Edges are labeled with identifiers (i.e., URIs in predicate position) that make them distinguishable from each other and allow for multiple edges between the same subject and object.

We can distinguish between predicates in given triples that form a relation or predicates that express an attribute:

➳ Definition 3: Property, Relation, Attribute

Let $\langle s, p, o \rangle$ be an RDF triple of RDF graph G.
The resource identified by p is called the property of the triple.
p is called a relation, if o is either a URI or a blank node $o \in (\mathcal{U} \times \mathcal{B})$.
p is called an attribute, if o is a literal $o \in \mathcal{L}$.

✎ Example 4

The RDF triple, published by ACME, that describes the statements "Metallica's place of origin is Los Angeles" is:

```
http://acme.example.org/api/metallica ,
http://acme.example.org/vocabulary#origin ,
http://acme.example.org/api/losAngeles
```

This triple encodes the relation between the entities Metallica and Los Angeles.

The RDF triple that describes the statements "Metallica was founded in 1981." is:

```
http://acme.example.org/api/metallica ,
http://acme.example.org/vocabulary#founded ,
1981
```

This triple expresses an attribute of the band Metallica.

Both triples together form a graph, which is graphically represented in Figure 1

RDF Data Model and Format

While RDF is a graph-based data model, there are several serialisation formats that can represent RDF graphs. Originally, the W3C proposed an XML-based serialisation [35] for the adoption by RDF data processing and management tools. Because of its readability we use the Terse RDF Triple Language (Turtle) serialisation [7] for the examples throughout the thesis. Turtle is a compact syntax for

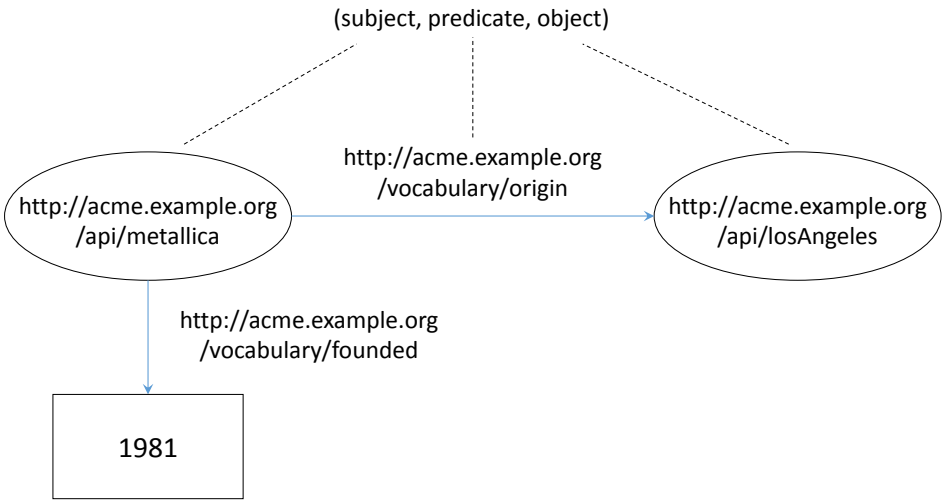


Figure 1: Example of an RDF graph (graphical representation).

RDF that allows for the representation of graphs with abbreviations for triples with the same subject or triples with same subject and predicate. Furthermore, the namespaces of URI can be abbreviated with prefix definitions[31]. Listing 1 and Listing 2 show the graph introduced in Example 4 in Turtle and RDF/XML serialisation respectively.

Listing 1: Turtle syntax representation of an RDF graph

```
@prefix acme: <http://acme.example.org/api/> .
@prefix p: <http://acme.example.org/vocabulary#> .

acme:metallica p:origin acme:losAngeles ;
        p:founded "1981" .
```

Listing 2: RDF/XML serialization of an RDF graph

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:p="http://acme.example.org/vocabulary#">

  <rdf:Description rdf:about="http://acme.example.org/api/metallica">
    <p:origin rdf:resource="http://acme.example.org/api/losAngeles"/>
    <p:founded>1981</p:founded>
  </rdf:Description>
</rdf:RDF>
```

The distinction between the actual data model and its serialisation is important with respect to the entropy an application faces, when interacting with Web resources. Both data model and format represent distinct sources of entropy that

[31] In the examples throughout the thesis we omit prefix definitions for brevity, if the namespace is clear due to the context of the example. For an overview of commonly used prefixes see http://prefix.cc/; retrieved 2015-04-10.

have to be considered when trying to negotiate the interoperability between a client application and an API (see Section 2.3.2). Staying within the RDF realm the data model is not affected by the choice of any of the serialization formats; the graph structures remain unchanged. Or from another point of view, a client application that supports the RDF data model must still handle the variety of the possible formats, i.e., being able to parse one or more of the available serialisations. However, the translation between the formats is merely a syntactical transformation. Client applications can leverage existing conversion tools[32] to transform retrieved RDF data to any required RDF format.

RDF Schema and Ontologies

RDF Schema (RDFS) is a vocabulary language for RDF [17] and allows the modeling of simple ontologies [104]. RDFS describes the logic dependencies among classes, properties, and values. While RDF provides universal means to encode facts about resources and their relationships, RDFS is used to express generic statements about sets of individuals (i.e., classes)[33]. With RDF it is already possible to specify the membership of an individual in a class, i.e., define instances of classes (by using the property `rdf:type`). RDFS allows to state the relations between classes and properties, thus providing more expressive semantics for the automated interpretation of data.

✎ **Example 5**

ACME provides schema information to help client applications to interpret the data about artists and events. Listing 3 shows an excerpt of the schema specification relevant to the RDF graph in Example 4.

The properties `p:origin` and `p:founded` are specified to be members of the class of all properties. `p:origin` is definied to be a subproperty of `p:hometown`, i.e., all relations between two nodes by `p:origin` also constitute a relation between the same nodes by `p:hometown`. By definition of domain and range, it is specified that `p:origin` always forms a relation between instances of the class `p:Artist` and instances of the class `p:City`. Similarly domain and range definitions specify that `founded` provides an attribute (i.e., a label) for instances of the class `p:Artist`. Finally the schema defines that `p:Artist` is a subclass of `p:Performer`, i.e., all instances of the class `p:Artist` are also instances of `p:Performer`.

The schema itself is also a Web resource, which can be retrieved by resolving e.g., the URI `p:origin`. Thus, a client that encounters RDF triples including the property `p:origin`, can simply resolve the URI to gain access to the schema to interpret the triples.

[32] see e.g., `http://www.easyrdf.org/converter`

[33] Both individual and classes are resources, in the sense of Definition 1

Listing 3: Specification of domain and range of properties in RDFS

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix p: <http://acme.example.org/vocabulary/> .

p:origin rdf:type rdfs:Property ;
        rdfs:subPropertyOf p:hometown ;
        rdfs:domain p:Artist ;
        rdfs:range p:City .

p:founded rdf:type rdfs:Property ;
        rdfs:domain p:Artist ;
        rdfs:range rdfs:Literal .

p:Artist rdfs:subClassOf p:Performer
```

Beyond RDFS more expressive entailment regimes are possible to enable more automatic inferences, e.g., by using the Web Ontology Language (OWL) [71] or fragments of OWL. The use of more expressive semantics implies a trade-off: A higher expressivity allows for more inferences, thus potentially enabling clients to draw inferences necessary to interpret data automatically to mitigate entropy. However, the expressivity is bought with increasing complexity resulting in longer processing times to draw inferences, i.e., increasing the challenge to reduce application runtime.

OWL LD combines features of RDFS and OWL with the goal to define adequate, implementable and robust semantics that cover many use-cases [39]. The entailment of OWL LD includes e.g., subclass and equivalence expressions, but relinquishes other OWL concepts, such as intersection of classes or cardinality expressions[34].

However, only a general automated Web agent has ultimately the requirement to be able to interpret a predefined set of entailment instructions. Given the trade-off between expressivity and complexity, an application for a specific task should be designed with only the features that result in the entailment of the inferences necessary to achieve the goal of the application; all features unnecessary to draw the required inferences should be excluded (see Chapter 3).

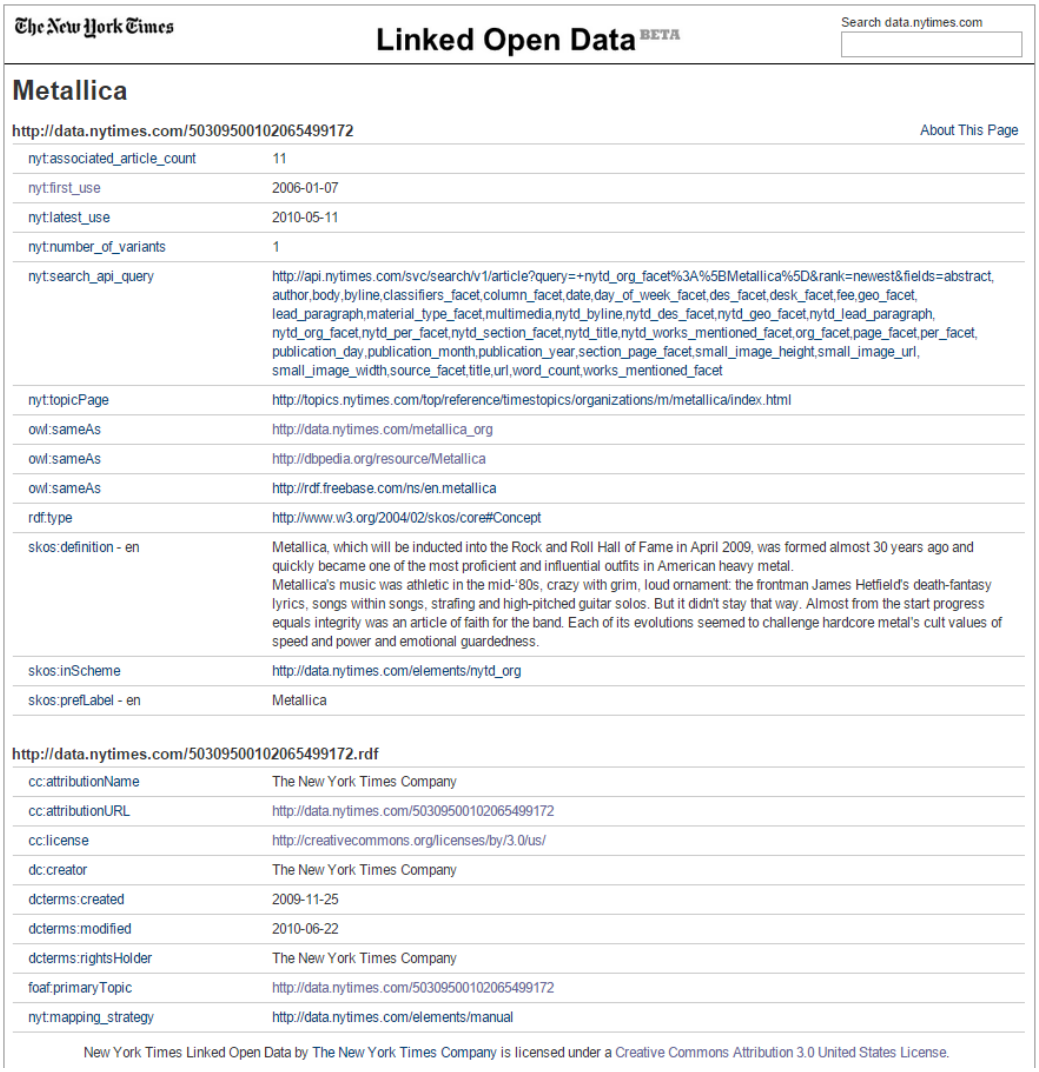
2.2.2 *Interlinked Resources*

The use of an HTTP URI allows machines and humans to lookup the name and get useful information about resources adhering to the RDF standards. The Hypertext Transfer Protocol (HTTP) is used to exchange data on the Web[35]. The use of an HTTP URI further guarantees the uniqueness of the identifier.

Linked Data identifies a subset of resources as information resources. Information resources are documents that actually contain RDF triples. Thus resolving the URI of an information resource returns triples directly, while resolving a

[34]For a complete overview see http://semanticweb.org/OWLLD/; retrieved 2015-04-10.

[35]See the IEEE RFC7230 specification for details at https://tools.ietf.org/html/rfc7230; retrieved 2015-04-10.



The New York Times — Linked Open Data BETA — Search data.nytimes.com

Metallica

http://data.nytimes.com/50309500102065499172 — About This Page

nyt:associated_article_count	11
nyt:first_use	2006-01-07
nyt:latest_use	2010-05-11
nyt:number_of_variants	1
nyt:search_api_query	http://api.nytimes.com/svc/search/v1/article?query=+nytd_org_facet%3A%5BMetallica%5D&rank=newest&fields=abstract, author,body,byline,classifiers_facet,column_facet,date,day_of_week_facet,des_facet,desk_facet,fee,geo_facet, lead_paragraph,material_type_facet,multimedia,nytd_byline,nytd_des_facet,nytd_geo_facet,nytd_lead_paragraph, nytd_org_facet,nytd_per_facet,nytd_section_facet,nytd_title,nytd_works_mentioned_facet,org_facet,page_facet,per_facet, publication_day,publication_month,publication_year,section_page_facet,small_image_height,small_image_url, small_image_width,source_facet,title,url,word_count,works_mentioned_facet
nyt:topicPage	http://topics.nytimes.com/top/reference/timestopics/organizations/m/metallica/index.html
owl:sameAs	http://data.nytimes.com/metallica_org
owl:sameAs	http://dbpedia.org/resource/Metallica
owl:sameAs	http://rdf.freebase.com/ns/en.metallica
rdf:type	http://www.w3.org/2004/02/skos/core#Concept
skos:definition - en	Metallica, which will be inducted into the Rock and Roll Hall of Fame in April 2009, was formed almost 30 years ago and quickly became one of the most proficient and influential outfits in American heavy metal. Metallica's music was athletic in the mid-'80s, crazy with grim, loud ornament: the frontman James Hetfield's death-fantasy lyrics, songs within songs, strafing and high-pitched guitar solos. But it didn't stay that way. Almost from the start progress equals integrity was an article of faith for the band. Each of its evolutions seemed to challenge hardcore metal's cult values of speed and power and emotional guardedness.
skos:inScheme	http://data.nytimes.com/elements/nytd_org
skos:prefLabel - en	Metallica

http://data.nytimes.com/50309500102065499172.rdf

cc:attributionName	The New York Times Company
cc:attributionURL	http://data.nytimes.com/50309500102065499172
cc:license	http://creativecommons.org/licenses/by/3.0/us/
dc:creator	The New York Times Company
dcterms:created	2009-11-25
dcterms:modified	2010-06-22
dcterms:rightsHolder	The New York Times Company
foaf:primaryTopic	http://data.nytimes.com/50309500102065499172
nyt:mapping_strategy	http://data.nytimes.com/elements/manual

Figure 2: Linked Data information resource provided by the New York Times (HTML representation) with statements about the document at the bottom.

resource that is another entity than an RDF document refers to an information resource that describes the entity (see Definition 1).

To have a URI for each, the resource representing an entity and the resource that is the document that describes the entity, allows to make a clear differentiation between statements about the entity and statements about the document. E.g., the Linked Data resource shown in Figure 2 (rendered in an HTML page[36]) contains statements about the band Metallica at the top, such as the number of associated articles, and statements about the document itself at the bottom, like licensing information.

The referral to an information resource when a client tries to resolve the URI of an entity that is not a RDF document can be realised in two ways:

- A server can redirect the client via HTTP status code `303 SEE OTHER` and provide the URI of the infomation resource that describes the entity. The client has to resolve the provided URI to access the data about the entity.

[36] http://data.nytimes.com/50309500102065499172; retrieved 2015-04-10.

- The URI of the entity can contain a fragment behind a hashtag #[37]. Following the HTTP specification a client is supposed to crop the fragment from the URI before resolving the URI. Thus the complete URI with fragment can be interpreted as the URI of the entity, while the URI without the fragment can be interpreted as the URI of the information resource.

The approach to use a hashtag-fragment is potentially faster, because it does not require to establish an additional HTTP connection as it is the case with the approach via redirects. However, a redirect is closer to the HTTP specification as it also allows for content negotiation, as described in Section 2.3.2.

The Linked Data principles require that the retrieved data, when resolving a URI, contains links to other resources so that clients can follow those links to discover further relevant information. We define links between resources in the context of Linked Data as follows:

➳ Definition 4: Information Resource, Link

A linked data information resource, identified by an HTTP URI u, is a document that contains a set of triples $\langle s, p, o\rangle$, denoted as T^u.
We denote $r \xrightarrow{i} T^u$, if a resource, identified by the HTTP URI r, refers to the triples of the information resource, identified by the HTTP URI u, i.e., u is the information resource of r. If r is an information resource itself, it holds true that $r \xrightarrow{i} T^r$.
There is a link from a resource r_1 to a resource r_2, denoted with $r_1 \xrightarrow{l} r_2$, if u is the information resource of r_1 and r_2 appears as subject or object in at least in one triple of T^u, i.e., $\exists t \in T^u : r_1 \xrightarrow{i} T^u \wedge (t = \langle r_2, p, o\rangle \in T^u \vee t = \langle s, p, r_2\rangle \in T^u)$

Links can be internal or external. An internal link is a link between two resources with the same namespace (i.e., both resources are from the same provider). An external link is a link to a resource with a different namespace, which can specifically be used to express relations between resources of different providers. In particular, the property `owl:sameAs` specifies the equivalence of two resources that represent the same thing.

External links are also used to link schemata of different providers, expressing relations between classes and properties in the different schemata. Overlapping information from different resources can be aligned using such equivalence statements and schema mappings.

✎ **Example 6**

ACME regularly includes external links in their provided resources to allow users to find additional information about events and performers. The band Metallica is identified by ACME with the URI

```
http://acme.example.org/api/metallica
```

[37] See the IEEE RFC3986 specification for details at https://www.ietf.org/rfc/rfc3986.txt; retrieved 2015-04-10.

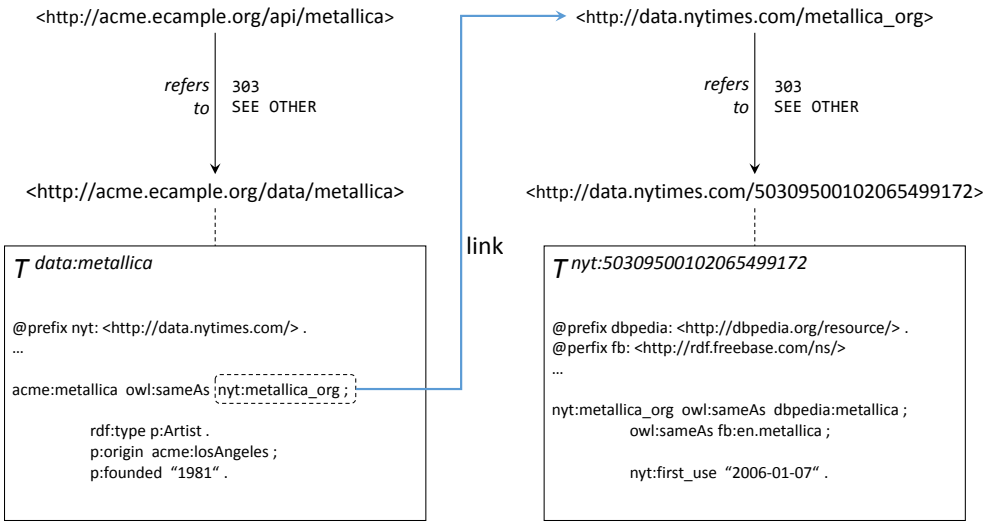


Figure 3: Illustration of an external link between two resources as equivalence relation. (Some prefix definitions omitted for brevity.)

When a client application resolves the URI the server refers the client to an information resource, as ACME cannot send the actual band to the client. The information resource is identified with the URI

```
http://acme.example.org/data/metallica
```

Resolving the information resource returns triples describing the band, including the following triple constituting a link to the Metallica resource provided by the New York Times as equivalence relation:

```
acme:metallica owl:sameAs nyt:metallica_org .
```

The resource provided by the New Your Times also refers to an information resource, which is also depicted in Figure 2. Here, further links to resources representing Metallica from other providers can be found. The link from the resource provided by ACME to the resource provided by the New York Times is illustrated in Figure 3.

2.2.3 *Basic Graph Pattern*

The established query language for RDF is SPARQL protocol and RDF query language (SPARQL) [123]. While SPARQL offers a wide range of features, Basic Graph Pattern (BGP) are an important fragment, which cover a wide range of information needs. We focus on BGPs as they are not only relevant for queries, but also form a major building block of the rule language described in Chapter 3 and Chapter 4.

➳ Definition 5: Triple Pattern, Basic Graph Pattern, Join Variable

Let $\mathcal{V}$ be the infinite set of variables, disjoint with $\mathcal{U}$,$\mathcal{B}$, and $\mathcal{L}$.
The union of all variables and RDF terms is denoted by $\mathcal{E} = \mathcal{V} \cup \mathcal{T}$.
A triple pattern, denoted with TP, is a triple $\langle s, p, o \rangle \in (\mathcal{V} \cup \mathcal{U} \cup \mathcal{B}) \times (\mathcal{V} \cup \mathcal{U}) \times (\mathcal{V} \cup \mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$, where subject, predicate and object can either be a variable or an RDF Term.
A basic graph pattern is a set of triple patterns $Q = \{t_1, ..., t_n\}$.
We denote $\mathcal{V}_Q$ as the finite set of all variables in the basic graph pattern Q.
We denote $\mathcal{E}_Q = \mathcal{V}_Q \cup \mathcal{T}_Q$ as the union of all variables and RDF terms in the basic graph pattern Q.
If two triple patterns in a basic graph pattern $t_k, t_l \in Q$ share the same variable $v_j \in \mathcal{E}_{\{t_k\}} \wedge v_j \in \mathcal{E}_{\{t_l\}}$, the variable v_j is the join variable for the triple patterns t_k and t_l in Q.

The triple patterns of a BGP form a graph similar to RDF triples. Intuitively to execute a BGP query Q over an RDF graph G implies finding a mapping of all variables in Q to RDF terms in G in such a way that the substitution of variables in Q with their mapped terms, yields a subgraph of G. Computing answers to a BGP query over an RDF graph amounts to the task of graph pattern matching:

➳ Definition 6: Solution Mapping, Instance Mapping, Result Binding

A solution mapping is a partially defined function $\mu : \mathcal{V} \to \mathcal{T}$ that maps variables to RDF terms. A solution sequence is a possibly unordered list of solution mappings.
We denote $dom(\mu) \subset \mathcal{V}$ as the domain of μ, i.e. the subset of $\mathcal{V}$ where μ is defined.
An RDF instance mapping is a partially defined function $\iota : \mathcal{B} \to \mathcal{T}$ that maps blank nodes to arbitrary RDF terms $t \in \mathcal{T}$.
Let Q be a BGP and G an RDF graph. Further let the pattern instance mapping $P^{\iota}_{\mu} : \mathcal{E}_Q \to \mathcal{T}$ be a function that maps variables and RDF terms in Q to RDF terms[a]:

$$P^{\iota}_{\mu}(x) = \begin{cases} \mu(x) & \text{if } x \in \mathcal{V} \\ \iota(x) & \text{if } x \in \mathcal{B} \\ x & \text{if } x \in \mathcal{U} \cup \mathcal{L} \end{cases}$$

We denote $P^{\iota}_{\mu}(\mathcal{E}_Q)$ for the RDF graph resulting from a substitution of all elements $e \in \mathcal{E}_Q$ in the BGP Q according to $P^{\iota}_{\mu}(e)$.
A mapping μ for the variables $\mathcal{V}_Q$ is a result binding for Q from G if $P^{\iota}_{\mu}(\mathcal{E}_Q)$ is a subgraph of G; i.e., μ satisfies that

$$\exists \iota \forall \langle s, p, o \rangle \in Q : \langle P(s), P(p), P(o) \rangle \in G \text{ and } dom(\mu) = \mathcal{V}_Q$$

We denote $\Omega_g(Q)$ for the set[b] of all unique result bindings for the BGP Q from RDF graph G, i.e.,

$$\Omega_g(Q) = \{\mu | \exists \iota : \ P^{\iota}_{\mu}(\mathcal{E}_Q) \subseteq G\}\}$$

[a] We deviate slightly from the SPARQL specification in http://www.w3.org/TR/sparql11-query/ by explicitly defining pattern instance mappings as function over RDF terms and variables. However, the semantics remain unchanged.

[b] Please note that we assume set semantics, in-line with [88, 97].

Finally we can define the join of two sets of solution mappings:

Definition 7: Compatible Mappings, Join

Two solution mappings μ_1 and μ_2 are compatible, if every join variable v_j is mapped to the same term, i.e., $\forall v \in (dom(\mu_1) \cap dom(\mu_2)) : \mu_1(v) = \mu_2(v)$. We denote $merge(\mu_1, \mu_2)$ as the union of two mappings, such that

$$merge(\mu_1, \mu_2) = \{v \mapsto \mu_1(v) | v \in dom(\mu_1)\} \cup \{v \mapsto \mu_2(v) | v \in dom(\mu_2)\}$$

If μ_1 and μ_2 are compatible, $merge(\mu_1, \mu_2)$ is also a mapping.
Let Ω_1 and Ω_2 be sets of solution mappings. The join of Ω_1 and Ω_2 is the set of all solution mappings resulting from the union of all mappings in Ω_1 and Ω_2 that are also mappings, i.e.,

$$join(\Omega_1, \Omega_2) = \{merge(\mu_1, \mu_2) \,|\, \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2,\ (\mu_1, \mu_2) \text{ compatible}\}$$

For a complete definition of the semantics of all features of SPARQL see [89]. We require the RDF instance mapping in Definition 6 to handle blank nodes in BGPs. Intuitively blank nodes act like variables, whose scope is outside of the result binding, i.e., variables that are not part of the solution mapping.

Data providers sometimes provide query capabilities, which can be remotely accessed via the Web. These query capabilities are referred to as SPARQL endpoints and allow for federated query processing. However, providing remote query capabilities represents a technical as well as an economic challenge for data providers as the endpoints must be managed. Consequently many providers restrict themselves to the Linked data principles and offer data simply as URI-identified resources, retrievable via HTTP without processing queries remotely on the servers of the provider.

Making data retrievable as Linked data resources prevents that the strain of all queries from a multitude of user clients is put on a central query processor maintained by the data provider, by pushing the effort of querying onto the clients. The clients consuming Linked Data need not execute their queries over the potentially large data dumps from a provider, as the resource-centric data model of Linked Data results in small encapsulated units of data. Clients can just retrieve a required subset of data by following the links between the resources.

Because links can also point to resources from third party providers, clients can assemble the data required for their task from a multitude of sources. To align the resources from providers, who might use different vocabularies one can draw inferences provided by schema mappings such as equivalence or subclass

statements between classes and instances, as explained in Section 2.2.1. SPARQL as defined by the W3C is a pure query language and does not stipulate any inferences over the processed data. In [38] an extension of SPARQL with the entailment regimes of RDF, RDFS and OWL is proposed. However, in a scenario where clients operate over Linked Data resources, clients can be developed, which only materialise the statements required for the goal of the client.

✎ **Example 7**

We consider an application that analyses how much time successful music bands on average need until they are widely known. For this purpose the application compares the founding date of bands with the time they are first mentioned in large newspapers. The required information for the analysis is gathered from Web resources describing the bands.

In particular the following BGP query is used in the application, based on the vocabulary employed by ACME and the New York Times:

```
?band  rdf:type  p:Artist .
?band  p:founded  ?date .
?band nyt:first_use ?first .
```

Intuitively the BGP queries for anything that is of type artist, its founding data and its first use in the New York Times. By default, the application retrieves the information resource describing a band from ACME. Because ACME does not offer information about the first publication in newspapers, the application follows links to acquire the corresponding data. For instance, in the case of the band Metallica the application follows the link as outlined in Example 6 and retrieves the resource from the New York Times.

The application has the following RDF graph available:

```
acme:metallica  rdf:type  p:Artist .
acme:metallica p:founded  "1981" .
acme:metallica p:origin acme:losAngeles .
acme:metallica owl:sameAs nyt:metallica_org .
nyt:metallica_org nyt:first_use "2006-01-07" .
```

However, the result binding for the query from this graph is empty, as there is no mapping for the join variable `?band`. Only if the semantics of the property `owl:sameAs` are taken into account, the equivalence of `acme:metallica` and `nyt:metallica_org` can be inferred and the following triple can be materialised and added to the graph:

```
acme:metallica nyt:first_use "2006-01-07" .
```

Now the query gives a result binding with the following solution mapping:

μ(?band) = acme:metallica
μ(?date) = "1981"
μ(?first) = "2006-01-07"

Thus the application acquires the necessary information to complete its task. Other inferences stemming from other entailment features, like the fact that p:Artist is a subclass of p:performer are not required for the application to achieve its goal.

2.3 REPRESENTATIONAL STATE TRANSFER

LD implies an interaction schema with resources based on HTTP, but LD is merely focused on the provisioning of data and data consumption by client applications. However, the tasks of Web-based applications often require services that go beyond the retrieval of resources. Examples for such services include the search and ordering of products, the booking of flights and hotels, or the upload of user generated content.

The term *Web Service* has several different definitions in the literature; for an overview see [19, 33]. In the context of this thesis we use term broadly to refer to any kind of system to provide functionality offered by a provider over the Web. While the offer of a service is Web-based the actual delivery of the functionality can entail not Web-based aspects and effects, e.g., the actual physical delivery of an online ordered product. For brevity we often use the term *service* when we refer to the functionality offered by a Web Service.

In this section we focus on Representational State Transfer (REST), a paradigm for the resource-oriented interaction with services on the Web. In particular we introduce REST by differentiating resource-oriented APIs from operation-oriented APIs, based on our work in [110].

An Application Programming Interfaces (API) is generally used to provide access to the functionality of a Web service. An application can use one or more APIs to implement Web-based functionalities and offer the services to users. If an application implements more than one service, the application has to integrate the services to make use of the combined functionality. The resulting layered architecture is depicted in Figure 4.

A Web site, can include pages containing interactive Web forms, which make use of the API to offer functionalities to users. Users have to use browsers to render and interact with the Web page. Thus, a browser together with the Web page build a client application for the API.

Client applications like mobile apps or web sites can be provided by the service providers themselves or by third parties. An API can be open or closed:

OPEN API: Third parties, different from the provider, can access and make use of the API. An API is also considered to be open, if the access is not free, i.e., the provider charges money for the use of the API, which allows to establish a whole business model around the provisioning of services on the Web. Open APIs provide the advantage that applications using the Web

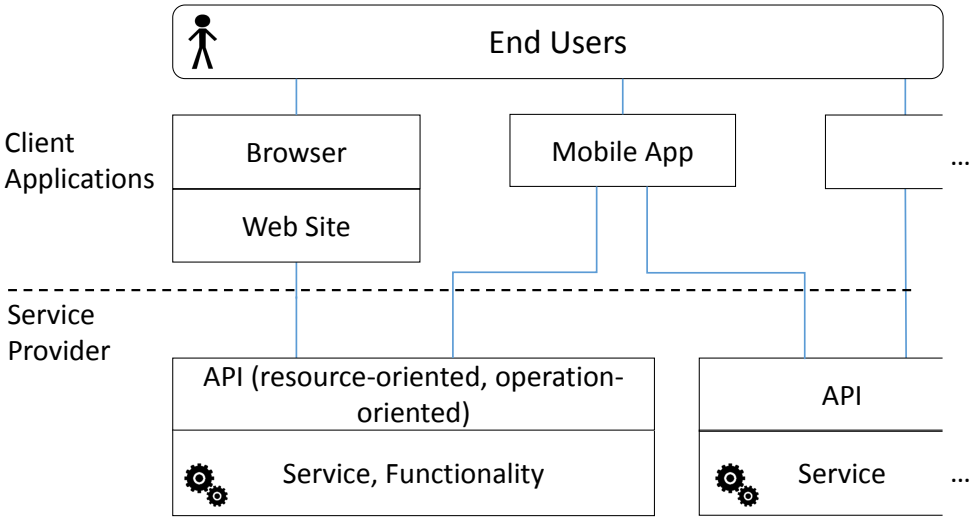


Figure 4: Architecture of services provided via APIs, implemented in client application for the consumption by end users.

API that are developed by third party developers increase the potential reach of the service, i.e., increase the amount of potential users.

CLOSED API: The API is only for the internal usage within the provider organisation. Closed APIs are useful to allow providers to group and combine functionalities to products. Either the functionality of the underlying service is only a supporting function for the business model of the provider, or the functionality is offered to end users only via client applications developed by the provider itself.

✎ **Example 8**

ACME offers an open API that allows to search for music artists and sports teams. Further tickets for upcoming concerts and matches of the artists and teams can be bought via the API. The tickets can be downloaded as digital version or are delivered physically.

The Web site of ACME makes use of the API to offer these functionalities to users. Further ACME offers a mobile app for the Google Android operating system as additional client application, so that users can spontaneously buy tickets en route. ACME did not invest into the development of an mobile app for the Apple iOS operating system. However, a third party developer offers an iOS mobile app, that allows users to buy tickets from ACME, but also from other ticket agencies, which offer APIs. The business model of the third party developer is based on advertisements shown in the mobile app.

According to a W3C note[38], when constructing a Web API two general alternatives for the implementation exist:

[38] http://www.w3.org/TR/ws-arch/#relwwwrest; retrieved 2015-04-10.

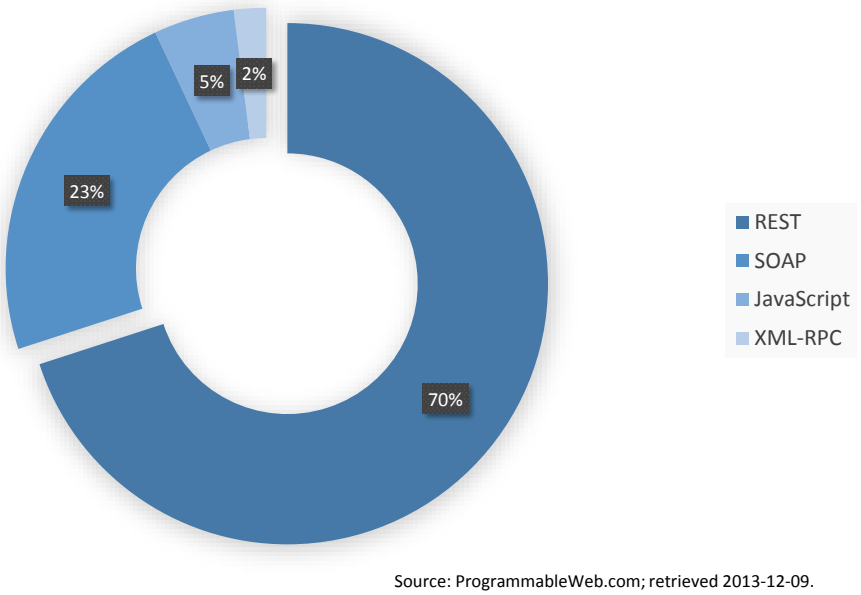


Figure 5: Proportion of different protocols used for open Web interfaces.

- An API that exposes an arbitrary set of operations.
- A resource-oriented API with a uniform set of stateless operations.

We give a brief overview of operations-oriented approach for differentiation and focus on the architecture of resource-oriented interfaces, since the latter are predominantly used on the Web (see Figure 5): A detailed comparison of both API paradigms is available from [82, 109].

2.3.1 *Operation-oriented API*

An API with arbitrary operations is often designed by adhering to the variety of specifications and languages commonly referred to as the WS-* stack. Arguably the most well known of these technologies are the Simple Object Access Protocol (SOAP) [43], which defines the XML-based exchange of structured information in distributed environments, and the Web Service Description Language (WSDL) [23] to describe service endpoints, which offer operations.

However, other variants of operation-oriented approaches exist, e.g., remote procedure calls. The main characteristic of this style of API is that the operations that make out the service are directly defined and offered. When implemented for services such an API uses the Web as a transport layer for the data and entails a high degree of freedom.

> ✎ **Example 9**
>
> ACME used to offer the functionalities described in Example 8 with an operation oriented approach. In particular the API offered the following

operations, which could be invoked by client applications for the corresponding functionality.

- `findArtistInfo` using the name of an artist as input.
- `findConcerts` using an identifier of concerts as input.
- `orderConcertTicket` using an identifier of the concert as input.

2.3.2 *Resource-oriented API*

A resource-oriented API complies to the constraints of a Representational State Transfer (REST) architecture [29]. REST is identified as the interaction between a client (i.e., an application) and a server based on three principles, according to the Richardson maturity model [94]:

1. The use of URI-identified resources.
2. The use of a constrained set of operations, i.e., the HTTP methods, to access and manipulate resource states.
3. The application of hypermedia controls, i.e., the data representing a resource contains links to other resources. Links allow a client to navigate from one resource to another during interaction.

Resource Structure

A REST API offers the service functionality under the primacy of resources rather than operations. A resource can be a real world object or a data object on the Web. Resources are uniquely identified with a URI (cf. Definition 1). In contrast to Linked Data REST does not make an explicit difference between a resource representing a real world object and a document containing data that describe the object [29] (cf. Definition 4). If a client resolves a URI that identifies a real world object, the data representation of the resource is returned directly to the client, without any referral. The representation of a resource details its current state, i.e., relevant information associated with the resource.

Resources can be grouped into collections. These collections in turn are URI-identified resources themselves and referred to as *collection resources*. The resulting tree-like structure of the resources is similar to the well known directory structure of file systems.

Constrained Operations Set

The interaction of client applications with services via a REST API is not based on the call of API-specific operations but rather on the direct manipulation of exposed resource representations or the creation of new resource representations. If a resource represents a real world object, a manipulation of the state representation might imply that the represented resource is manipulated accordingly, i.e, there can be an implicit connection between the data representation of a resource and the actual object, that is represented. This connection guarantees that

the data representation of a resource always conforms to the actual state of the represented entity. The implication of the such a connection is that

- if the data representation is manipulated, the represented entity has to change accordingly;
- if the represented entity changes its state, the data representation has to change as well to reflect the new state.

However, the connection between a data representation and the represented object is not always given. E.g., if the origin city of the band Metallica is changed in the data representation, the actual origin of the band stays the same (see Section 4.2).

To manipulate data representations, REST offers only a constrained set of operations that can be applied to a resource. These operations are shared by all interfaces following REST principles, thus the entropy for clients interacting with REST APIs is significantly reduced. The constrained set of operations increases interoperability and understandability of the interfaces. Nevertheless, not all resources must necessarily allow the application of all possible methods.

Some of the methods can carry input data as a payload, which describes the intended new state of the addressed resource[39]. On the Web, the allowed operations are the HTTP methods[40] (see Table 2).

After using a method to interact with a resource the client application receives a status code in the response, which informs the client about the success or failure to which the application can react at run time. The status codes can be categorised as follows:

1XX (INFORMATIONAL) Indicates a provisional response to a request, prior to the regular final response, e.g., used to inform a client that it can continue with its request.

2XX (SUCCESSFUL) Indicates that the request was understood and accepted,e.g., used to inform the client that a resource has been created.

3XX (REDIRECTION) Indicates that further action by the client is required to complete the request, e.g., used to inform the client that the request should be redirected to another resource.

4XX (CLIENT ERROR) Indicates that the request failed and the fault lies with the client, e.g., used if the requested resource does not exist.

5XX (SERVER ERROR) Indicates that the request failed and the fault lies with the server, e.g., used if a service is not available.

Two types of methods exist: safe and unsafe. Safe methods guarantee not to affect the current states of resources, while unsafe methods change the state of the resources. Furthermore, most of the methods are idempotent. The repeated

[39]The HTTP POST method is a noteworthy exception as it permits the submission of data to process, which is similar to an RPC call and therefore should be used carefully.

[40]See the IEEE RFC7231 specification for details at https://tools.ietf.org/html/rfc7231; retrieved 2015-04-10.

Table 2: Overview of HTTP methods (excerpt) and their characteristics.

Method	Safe	Idempotent	Description
GET	✓	✓	Retrieve the current state of a resource.
OPTIONS	✓	✓	Retrieve a description of possible interactions.
DELETE		✓	Delete a resource.
PUT		✓	Create or overwrite a resource with the submitted input.
POST			Send input as subordinate to a resource or submit input to a data-handling process.

application of an idempotent method on a resource does not change the state of the resource beyond the first application of the method. For example, if a client application deletes a resource, deleting it again has no effect.

The application of a HTTP method is called a request to which a server replies with a response. We define request and response as follows[41]:

Definition 8: Request, Lookup, Response

Let $M = \{GET, PUT, POST, DELETE, OPTIONS\}$ be the finite set of HTTP methods.
Let C be the finite set of HTTP status codes.
A request is a tuple $req = (m, u, D^{req})$ with

- $m \in M$, an HTTP method
- $u \in \mathcal{U}$, the URI of the addressed resource
- D_{req}, a dataset as request payload

A lookup is a request with $m = GET$ and $D^{req} = \emptyset$
A response is a tuple $res = (c, D^{res})$ with

- $c \in C$, a HTTP status code
- D^{res}, dataset as response payload

Specifically, we denote $\beth(req)$ for the response to the request req.
We denote $RP(req)$ for the dataset returned in the response to request req, i.e.,

$$\beth(req) = (c, RP(req))$$

Hypermedia Controls

A REST API fosters loose coupling between clients and services on the premise that client applications do not need to know about all available resources in advance. The retrievable representations of some known resources contain links

[41] For simplicity we constrict the definition to the attributes relevant for the approaches in the following chapters and exclude most of the header information potentially contained in HTTP Requests and Responses.

to other resources that the client can discover during runtime. Applications can use such discovered resources to perform further interaction steps. Collection resources specifically contain links to all the resources in the collection. This architectural design allows client applications to be robust toward changes in the API, because the client application has to react to whatever the client finds when it interacts with the API [94, 124].

In conjunction with the constrained set of available operations, the approach to have client applications react to the state of a resource as discovered at runtime shifts the entropy from the execution of service operations to the interpretation of the data representations of the resources. Intuitively it is easy for a client application to choose the right operation for a task (little entropy), but hard to interpret the meaning of the data representing a resource and the contained links (large entropy).

In a REST architecture, no constraints are given on how the state of a resource has to be represented. There is no defined standard regarding data model or serialization format of the data that detail the current state of a resource or the input and output data of a method.

Client application and API are, however, supposed to agree on the format of the exchanged data and implicitly on how the data is supposed to be interpreted. The process of establishing this agreement is called *content negotiation*. For different application scenarios such an agreement requires vendor specific content types (i.e., content types defined by the service provider) for the individual services to convey the meaning of the communicated data.

If a client application applies a method to a resource requesting a certain content type, the client signals, that it is capable to parse the corresponding data format and interpret the data following the intention of the provider. A server that is not able to provide the requested content type has to deny the application of the method (response code 406 NOT ACCEPTABLE).

The idea behind vendor specific content types is that service providers can reuse content types and application developers can make use of specific content type processors in their applications to work with the data. Such content type processors would encapsulate the capabilities to interpret the resources from different providers, thus mitigating the entropy clients have to face interacting with resources.

In practice, however, most Web API providers simply make use of standard non-specific content types, e.g., `text/xml` or `application/json` [68]. Developers therefore have to write applications that are individually adapted for the API they make use of. In Chaper 4 we propose how the combination of Linked Data resource representations and REST can improve on this situation.

> ✎ **Example 10**
>
> The current API of ACME is resource-oriented. In particular ACME's API contains collection resources representing artists, teams, concerts, events, etc., which correspond to the folders of a file system. The resources representing the concrete instances of artists, teams, concerts and events correspond to files put into the corresponding folders. The API of ACME com-

prises thousands of resources, with the exact number constantly changing. The following list contains examples of instances and collection resources of the ACME Web API:

- Collection resources of music artists a sports teams respectively; the representation contains links to all artists and teams.
  - `acme:artistCollection`
  - `acme:teamCollection`
- Resources representing individual music artists; the representation contains information about the artist (e.g., name and origin) and links to upcoming concerts.
  - `acme:metallica`
  - `acme:eminem`
- Collection resource of concerts and matches; the representation contains links to all upcoming concerts and matches.
  - `acme:concertCollection`
  - `acme:matchCollection`
- Resources representing individual concerts; the representation contains information about the concert (e.g., location, date, and price per ticket), links to the performing artist and to an instance of the `acme:ticketorder` resource.
  - `acme:concert1234`
  - `acme:concert1235`
- Collection resource of ticket orderings; the representation contains links to all ticket orderings.
  - `acme:ticketorder`
- Resources representing individual ticket orderings with information about the ordering (e.g., delivery and address) and link to corresponding concert or match.
  - `acme:order567`
  - `acme:order568`

ACME allows HTTP operations on the resources in the following manner:

GET Almost all resources of the ACME API allow for the application of GET to retrieve information. A GET on one of the collection resources gives an overview of the known instances (i.e., concerts, matches, teams, and artists). The retrieval of the information of a specific ticket order, however, is only allowed with the correct credentials. Only a user who created the ordering can look it up again. The retrieval of the collection resource for all orderings is generally not permitted.

POST The collection resource for ticket orderings allows POST to enable users to add a new ticket order for a specific concert.

PUT The use of PUT is only allowed to overwrite existing ticket orders, thus enabling users to update an order.

DELETE Ticket orders can also be canceled with DELETE up to a predefined time before the corresponding concert or match takes place.

At the behest of an end user a client application can interact with the resources of the ACME API in the following way, leveraging hypermedia controls:

1. Retrieve the information (i.e., representation) of a specific artist, which contains links to upcoming concerts (method GET).
2. Retrieve the information of one of the concerts, which contains a link to the collection of ticket orderings (method GET).
3. Create in this collection a new resource representing a new ticket order (method POST).

For the described interaction only the identifier of the resource representing the artist needs to be known in advance (i.e., before the interaction can start). This could be further refined by providing a search interface over the available artists as contained in the artist collection resource.

3

PARALLEL PROCESSING OF WEB RESOURCES

3.1 INTRODUCTION

Linked Data has resulted in a large amount of Web resources openly available on the Web, providing information across a multitude of domains. Recent crawls of these Web resources confirm at least 4 Billion statements[42], and a total of 30 Billion statements[43,44] is estimated to be accessible.

Application clients can access the available information, combine, process the data and present the results to users. In particular, value can be generated when multiple data sources are integrated, as the combination of information obtained from multiple providers in large distributed environments such as the Web can result in new insights, which can not be obtained from a single source.

Linked Data supports the combination of resources from multiple sources by client applications as the Linked Data principles reduce the entropy encountered by the clients. The principles imposed by Linked Data mandate a uniform data model for the representation of resources as well as a way to access these resources with HTTP as access protocol [9].

Mappings between resources via hyperlinks lead to a very large interconnected data graph. But client applications built upon this data graph still face the problem of entropy, due to the fact that various sources use different identifiers to denote the same schema elements. The providers of Linked Data Web resources can use schemata with constructs from RDFS and OWL to help clients to interpret the resources. As different providers employ different vocabularies, links between the schemata establish relations between the different identifiers of objects and documents. Client applications can use these relations to align Web resources, i.e., integrate the data representing the resources by reasoning over the semantics of the data. Thus the client application is able to extract the required information for their task by evaluating queries over the integrated data.

However, it remains difficult to evaluate queries over data that is scattered across many sources and thus enabling client applications that perform with short runtimes in an environment, which can be characterised as follows:.

- The retrieval of required Web resources can be time consuming, as it is subject to network latency and available bandwidth and can stall the actual processing and query evaluation. The processing and retrieval of data has

[42] http://km.aifb.kit.edu/projects/btc-2014/; retrieved 2015-04-10.
[43] http://www4.wiwiss.fu-berlin.de/lodcloud/state/; retrieved 2015-04-10.
[44] http://wp.sigmod.org/?p=786; retrieved 2015-04-10.

to be carefully orchestrated, in particular because Linked Data implies the acquisition of the data based on the traversal of links between resources, which are dynamically discovered at runtime.

- To retrieve all available resources (or a very large subset, e.g., every resource in a specific domain) is prohibitively expensive. Client applications cannot follow all available links blindly, but should be designed in manner to specifically target individual resources and schemata.
- Taking into account all semantic features of data items for reasoning (such as transitivity, symmetry or equality of properties) is computationally expensive and often not needed for specific applications. Developers should be able to choose which features their application is going to support in such a way, that the application can evaluate the semantics of schemata, which are not known a priori and are retrieved at runtime.

In this chapter we join two currently isolated strands of research to provide query processing capabilities over Web sources:

- Methods for evaluating queries over interlinked sources via link traversal.
- Approaches for integrating data over interlinked schemata via reasoning.

We describe algorithms for the parallel streaming evaluation of cyclic (recursive) pipelined query plans in conjunction with network requests based on a data-driven execution model. Thus we present methods that intertwine the evaluation of query plans with network requests to fetch data. In particular, we focus in this chapter on retrieval and processing of Web resources, before we look beyond the consumption and address the manipulation of Web resourced in Chapter 4.

Building applications that use distributed Web resources to acquire and query relevant information involves several steps:

1. The Web resources have to be accessed and downloaded.
2. The retrieved data has to be integrated.
3. The integrated data has to be queried.

To access, integrate and query Linked Data multiple specialised systems can be employed. The use of multiple systems implies that the steps are carried out sequentially. Alternatively, one has to implement the data processing in imperative code; those mash-ups are often tailored for specific data sources in a narrowly-defined domain. We describe the architecture of a fully implemented system that is able to follow links in processed sources, retrieve the data from the newly discovered resources, integrates the data by reasoning over dynamically retrieved schemata, and evaluate queries , all in a cohesive integrated process.

We design and study methods to access and integrate data from disparate interlinked Web Resources. In our approach, the different steps can be encoded in a high-level specification based on rules. The topic of this chapter are methods for efficiently evaluating queries taking into account such specifications on shared memory machines.

Several systems exist that evaluate queries directly over resources accessible as Linked Data [48, 46, 30, 62]. However, these systems do not take the semantics of data sources (the mappings of schema and instance elements) into account during query processing. Other systems rely on a fixed set of sources, with fixed reasoning constructs. Reasoners [100, 76] operate over locally accessible (or a priory downloaded) single-source datasets. We assume a hyperlinked environment, in which applications benefit from an approach, where data access and data processing are interleaved.

Dataspace systems [32] rely on a centralised catalogue of sources and operate therefore in scenarios with less entropy. On the Web, we have to handle the links between resources at runtime to discover new data sources.

Stream reasoning systems [70] and complex event processing systems [5] rely on a fixed number of sources that push data. On the Web, polling is the prevalent communication mode, which is also fostered by Linked Data. Polling allows client applications to retrieve data when needed [58] and to include new reasoning constructs while traversing the graph of resources.

3.1.1 *Challenges*

We address the following challenges in the context of runtime requirements and entropy (see Section 1.1.1):

- To retrieve all available resource (or a very large subset, e.g., every resource in a specific domain) is prohibitively expensive. Client applications cannot follow all available links blindly, but should be designed in manner to specifically target individual resources and schemata. Further, there is little coordination between providers, and resources have to be aligned and integrated. The reasoning constructs required to align the resources can vary depending on the goal of the application. The supported degree of complexity for drawing inferences directly translates into an increased processing time. Rather than generally supporting all semantic features of an entailment regime, a developer should be able to chose which constructs to employ or to relinquish.
- There is little or no a priori information about the size or schema of the overall graph, if client applications collect information by traversing links, which are found piece by piece at runtime. Consequently, an approach to process the data cannot rely on traditional indexing architectures, in which the data is indexed before processing time. Queries over Linked Data have be executed directly over the resources dereferenced at query time. Thus, an execution model for programs to interact with Web resources has to achieve short processing times in the face of dynamic network lookups.
- Sources exhibit heterogeneous performance characteristics, which together with bandwidth and network latency constraints influence the time necessary to retrieve resources. An application client has to accommodate for the retrieval time as well as the processing time. Specifically, we require parallel algorithms, to deal with both the computationally expensive rule processing and the low bandwidth and unpredictable network latency of sources.

In the implementation of such algorithms different parallel tasks have to be coordinated and processing bottlenecks have to be avoided. Specifically, if the evaluation of interdependent rules and queries is conducted with parallel processes, it is hard to determine when the processing is finished, i.e., a fixpoint in the processing is reached.

3.1.2 *Contributions*

Our contributions are as follows:

- We describe rule-based programs that can be used encode reasoning features intertwined with link traversal specifications. In particular, we introduce the notion of request rules, which infer required network lookups from the processed data. Request rules complement deduction rules, which infer and materialise implicit information from the processed data (Section 3.2).
- We introduce a parallel execution model for the evaluation of queries and rules, in which operators of a physical evaluation plan schedule each other in a data-driven manner: every operator pushes intermediate results to subsequent operators in the evaluation plan within a single thread. Thus several sequences of operators can be executed in parallel. The scheduling of our parallel execution model avoids overhead from inter-process communication and thread scheduling of the operating system. At the same time the push approach caters to scenarios with dynamic network requests, where data from multiple sources has to be processed on arrival (Section 3.3).
- We describe several threading models to implement the execution model for the parallel evaluation of programs. We identify the different trade-offs of the individual threading models. In particular we show the details of a threading model that allows to use multiple queues for the input data of retrieved resources. The use of multiple input queues specifically eliminates the bottleneck that a single input queue otherwise represents. We describe for each of the identified threading models how to probe for the termination of the evaluation, i.e., when a fixpoint is reached. (Section 3.4.1). Additionally we detail an algorithm to coordinate the parallel processes and actively track when a fixpoint is reached, thus potentially avoiding unnecessary waiting time to finish the processing (Section 3.4.2). Our approach intertwines rule based reasoning and query processing with the network retrieval of resources. At the same time it allows to separate the network-related workload from the processing-related workload. The separation of workload allows to balance available computing resources between data processing and network lookups to minimise overall runtime depending on the application scenario (Section 3.4.3).

We describe experiments in Section 3.5, cover related work in Section 3.6, and conclude the chapter with Section 3.7.

3.2 RULE-BASED PROGRAMS

For a Web-based application to perform its intended task, it has to retrieve required resources, integrate the retrieved data by materialising implicit statements derived from the semantics of the data and evaluate queries over the resulting graph. In our approach we propose to specify the application logic with rules, which use BGPs as basis. In particular, we employ rule-based reasoning to align the vocabularies of the Web resources. The rules to encode the desired reasoning constructs are called *deduction rules*.

Definition 9: Deduction Rule

A deduction rule $\rho^d : \{B\} \implies \{H\}$ consists of two BGPs:

- the rule body B
- the rule head H

Every variable in the head H has also to be part of body B, i.e., $\mathcal{V}_H \subset \mathcal{V}_B$.
An execution step of a deduction rule ρ^d over a graph G adds the graph G_{add} to G, where G_{add} is the RDF graph resulting from the substitution of the variables in H according to all result bindings of B from graph G, i.e., $\forall \mu \in \Omega_G(B) \forall \langle s, p, o \rangle \in H : \langle P^\iota_\mu(s), P^\iota_\mu(p), P^\iota_\mu(o) \rangle \in G_{add}$ with a unique blank node mapping ι for every $\mu \in \Omega_G(B)$.
We denote f^d as the function that maps a deduction rule and a graph to the graph of derived triples resulting from one execution step of the deduction rule and $step^d$ as the function mapping to the resulting graph of an execution step:

$$f^d(\rho^d, G) = G_{add}$$

$$step^d(\rho^d, G) = G \dot{\cup} f^d(\rho^d, G)$$

The complete execution of a deduction rule ρ^d over graph G is the recursive evaluation of n execution steps of ρ^d over G until no new triples can be derived, i.e., a fixpoint is reached:

$$min(n) : (step^d_1 \circ ... \circ step^d_n)(\rho^d, G) = (step^d_1 \circ ... \circ step^d_{n+1})(\rho^d, G)$$

Deduction rules can be used to specify application specific data transformations and domain knowledge. Furthermore different rule sets encode reasoning constructs according to RDF, RDFS and OWL semantics. In particular, it is possible to employ only parts of the rule sets to restrict the reasoning to subsets of the entailment regimes. Thus, the rule-based processing allows to scale the employed reasoning features with respect to the requirements of the application.

Furthermore we also use rules to encode the requests to be performed according to the application logic, which are called *request rules*. In the context of this chapter we only focus on lookups as requests:

Definition 10: Request Rule

A request rule $\rho^r : \{B\} \implies \{x\}$ consists of

- a BGP B as rule body
- a variable or URI $x \in \mathcal{V} \cup \mathcal{U}$ as rule head.

If x is a variable $x \in \mathcal{V}$, then it has to appear in the rule body, i.e., $x \in \mathcal{V}_B$. The execution step of a request rule ρ^r over a graph G adds the graph G_{resp} to G, where G_{resp} is the RDF graph returned in the payload of resource lookups. The resources to lookup are identified by

- x, if B has at least one result binding from graph G, and if x is a URI.
- the URIs $\mu(x) \in \mathcal{U}$, to which x is mapped in all result bindings of B from G, if x is a variable.

The responses to the lookups[a] are defined by

$$(c, D^{res}) = \begin{cases} \mathfrak{I}((GET, x, \emptyset)) & \text{if } x \in \mathcal{U} \wedge \Omega_G(B) \neq \emptyset \\ \mathfrak{I}((GET, \mu(x), \emptyset)) \quad \forall \mu \in \Omega_G(B) & \text{if } x \in \mathcal{V} \wedge \mu(x) \in \mathcal{U} \end{cases}$$

Therefore the graph G_{resp} is defined as

$$G_{resp} = \begin{cases} RP(x) & \text{if } x \in \mathcal{U} \wedge \Omega_G(B) \neq \emptyset \\ \dot{\bigcup}_{\mu \in \Omega_G(B)} RP(\mu(x)) & \text{if } x \in \mathcal{V} \wedge \mu(x) \in \mathcal{U} \\ \emptyset & \text{otherwise} \end{cases}$$

We denote f^r as the function that maps a rule and a graph to the graph of retrieved triples resulting from one execution step of the request rule and $step^r$ as the function mapping to the resulting graph of an execution step:

$$f^r(\rho^r, G) = G_{resp}$$

$$step^r(\rho^r, G) = G \dot{\cup} f^r(\rho^r, G)$$

The complete execution of a request rule ρ^r over graph G is the recursive evaluation of n execution steps of ρ^r over G until no new lookups can be found, i.e. a fixpoint is reached:

$$min(n) : (step_1^r \circ ... \circ step_n^r)(\rho^r, G) = (step_1^r \circ ... \circ step_{n+1}^r)(\rho^r, G)$$

[a] For brevity we implicitly assume that all lookups are automatically referred to an information resource.

Intuitively a request rule allows to specify which resources to retrieve. The resources are identified in one of two ways:

- The URI of the resource is directly provided. If a match for the rule body is found in the processed graph, the identified resource is retrieved. To directly specify the resource allows to include resources in the processing, which are known to be useful a priori, but potentially are not interlinked with other processed resources.

- A variable in the rule body is identified. For every identified result binding of the rule body from the processed graph, the provided variable signifies a resource to lookup. If the mapping for the identified variable of a result binding points to a URI, the URI is the identifier of a resource to retrieve. The mapping of the provided variable could also point to a literal or a blank node, in which case we do not retrieve a resource. To determine the URIs dynamically from the processed data, allows to follow and expand the links between the resources.

Similar to deduction rules, request rules are applied recursively to a graph, where the retrieved triples are added to the graph after every step. The recursive application is necessary, because the retrieved data in one step can lead to the identification of further links to follow.

Request rules allow to determine in a fine grained manner what resources to retrieve and which resources to follow. Thus an application does not have to follow all links in an exploratory manner. Consequently, the results of employed queries are not necessarily complete in the sense that all by link traversal potentially reachable results are identified. However, request rules can be defined to follow all found links, thus achieving seed complete results. For an introduction into completeness classes of query results from Linked Data see [45]. The intuition behind the approach to request rules is, that the application logic can be specified in such a way that only necessary results are obtained rather then all possible results.

To define the application logic we combine deduction rules and request rules to *linked programs*:

➳ Definition 11: Linked Program

A linked program $\mathcal{P} = (G, R, P^d, P^r)$ is a tuple with

- G a finite set of initial triples, i.e., the starting graph,
- R a finite set of initial resources,
- P^d a finite set of deduction rules,
- P^r a finite set of request rules,

where $G \neq \emptyset \vee R \neq \emptyset$, i.e., there has to be at least one triple in the starting graph or one initial resource specified.
The execution of a linked program implies that the initial resources are retrieved and the returned triples are added to the starting graph G. The rules P^d and P^r are recursively executed over G until the fixpoint is reached. We denote $step^p$ as the function mapping to the resulting graph of an execution step of multiple deduction and request rules over a graph G:

$$step^p(P^d, P^r, G) = G \dot{\cup} \dot{\bigcup}_{\rho^d \in P^d} f^d(\rho^d, G) \dot{\cup} \dot{\bigcup}_{\rho^r \in P^r} f^r(\rho^r, G)$$

The complete execution of a program $\mathcal{P}$ is the recursive evaluation of n execution steps of all rules $P^d \in \mathcal{P}$ and $P^r \in \mathcal{P}$ until a fixpoint is reached:

$$min(n) : (step_1^p \circ ... \circ step_n^p)(P^d, P^r, G) = (step_1^p \circ ... \circ step_{n+1}^p)(P^d, P^r, G)$$

We denote $G_{\mathcal{P}}$ as the graph resulting from the execution of the linked program $\mathcal{P}$.
We denote $R_{\mathcal{P}}$ as the set of all resources retrieved by a linked program, which includes initial resources as well as resources retrieved via request rules.
BGP queries can be registered to linked programs. If a BGP query Q is registered to a program $\mathcal{P}$, the query Q is evaluated over the result graph $G_{\mathcal{P}}$ of program $\mathcal{P}$. We denote $\Omega_{\mathcal{P}}(Q)$ as the result bindings of query Q from program $\mathcal{P}$.

The evaluation of a linked program continuously extends the initial graph G. Specifically, both deduction and request rules monotonously add triples to G and are not capable of removing triples. Neither deduction rules nor request rules can generate new RDF terms, i.e., terms that are not present in the initial graph G, the retrieved resources $R_{\mathcal{P}}$, or the rules P^d and P^r. Consequently, given that the set of retrieved resources $R_{\mathcal{P}}$ is finite, a linked program is guaranteed to have a fixpoint, as the number of combinations of RDF terms to form valid triples is finite. Therefore, eventually a linked program will not be able to infer new triples in another recursion to add to the result graph, and reach the fixpoint. Consequently, the result graph will also be a finite set of triples.

However, on the Web we cannot guarantee that the set of retrieved resources is finite. Consider a set of resources, where every resource represents a natural number[45]. Retrieving a number yields a link to its next natural successor[46]. Now consider a linked program acting upon the resources of natural numbers, consisting of

- A request rule that specifies that for every found natural number, the successor of the number has to be retrieved,
- The URI of an arbitrary natural number in the initial requests.

The described program would never reach a fixpoint, as it would always find another resource (i.e., the next successor of a natural number) to retrieve.

In Section 3.3 we describe the architecture of a system to execute linked programs, which produces results of queries from programs incrementally, i.e., result bindings are returned as soon as they are found during execution of the program. Thus, an application can leverage the extracted information as early as possible, which reduces the runtime requirements of the application.

For the serialisation of the queries we use SPARQL syntax restricted to BGP queries. To serialise the rules of a linked program we use Notation 3 (N3) syntax, which is similar to Turtle, but allows for variables and triple quoting [11]. In particular, the allowed constructs in Turtle are a subset of the allowed constructs in N3. Listing 4 shows three deduction rules encoding the symmetry and transitivity of `owl:sameAs`, as well as the semantics of `owl:inverseOf`, to specify inverse properties.

[45] http://km.aifb.kit.edu/projects/numbers/; retrieved 2015-04-10.

[46] As the dataset of all natural numbers would be infinitely large, every resource has to be generated by the server upon request, rather than stored.

Listing 4: Deduction rules specifying the symmetry and transitivity of owl:sameAs and semantics of owl:inverseOf in N3 syntax.

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .

# (1) Symmetry of owl:sameAs
{ ?x owl:sameAs ?y . } => { ?y owl:sameAs ?x . } .

# (2) Transitivity of owl:sameAs
{ ?x owl:sameAs ?y .
  ?y owl:sameAs ?z . } => { ?x owl:sameAs ?z . } .

# (3) Inverse property
{ ?p1 owl:inverseOf ?p2 .
  ?x ?p1 ?y .             } => { ?y ?p2 ?x . } .
```

The implication symbol => is a short hand for the URI `log:implies`[47]. Therefore a rule in N3 is also a triple $\langle s, p, o \rangle$, with

- $s = B$; subject is the body of the rule.
- $p =$`log:implies`; predicate is the URI identifying the concept of implication.
- $o = H$; object is the head of the rule.

Request rules are encoded using W3C vocabularies[48,49] to describe HTTP messages and methods. Listing 5 shows two request rules, which specify that explicitly the schema of property `foaf:depiction` has to be retrieved, if a triple is found that uses the property as predicate, and that links of `owl:sameAs` relations should be followed.

Listing 5: Request rules specifying the explicit retrieval of a resource, and the following of all links established by owl:sameAs.

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix acme: <http://acme.example.org/api/> .
@prefix p: <http://acme.example.org/vocabulary#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix http: <http://acme.example.org/api/> .
@prefix httpm: <http://acme.example.org/vocabulary#> .

# (1) Explicit retrieval of foaf schema
{ ?x foaf:depiction ?y . } => { [] http:mthd httpm:GET;
                                   http:requestURI foaf:depiction .} .

# (2) Follow all owl:sameAs links
{ ?x owl:sameAs ?y . } => { [] http:mthd httpm:GET;
                               http:requestURI ?y .} .
```

[47] http://www.w3.org/2000/10/swap/log#implies; retrieved 2015-04.10.
[48] http://www.w3.org/2011/http; retrieved 2015-04.10.
[49] http://www.w3.org/2011/http-methods; retrieved 2015-04.10.

Our system treats the appearance of a predicate from the W3C HTTP vocabularies as keyword to differentiate between request and deduction rules. Intuitively the triples in the rule head are not directly derived, if results for the rule body are found, but simply identify resources to retrieve with the object of `http:requestURI`.

✎ **Example 11**

A third-party provider develops a Web application on which users can buy tickets to soccer games. One of the design features of the application is that the users are shown pictures of the stadium in which a game takes place.

ACME identifies the Allianz Arena in Munich as resource with the URI `acme:allianzArena`, which contains a relation to the location of the stadium, a link to the representation of the stadium from dbpedia[a], and a link to a picture of the stadium. The relation to the picture is established with the friend-of-a-friend vocabulary (foaf)[b]. Retrieving the resource returns the following graph:

```
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

acme:allianzArena p:location acme:munich ;
                  owl:sameAs dbpedia:Allianz_Arena .
acme:allianzArenaPic.jpg foaf:depicts acme:allianzArena .
```

The third party provider uses a linked program to retrieve pictures of stadiums. In particular to retrieve pictures of the Allianz Arena a linked program $\mathcal{P}$ can be used with

- $G = \emptyset$; an empty starting graph;
- $R = \{$`acme:allianzArena`$\}$; the resource of the stadium provided by ACME as initial resource;
- P^d: the deduction rules as shown in Listing 4;
- P^r: the request rules as shown in Listing 5.

To extract the URIs of pictures the following query is registered to $\mathcal{P}$:

```
SELECT ?x
WHERE {?x foaf:depicts acme:allianzArena . }
```

The execution of $\mathcal{P}$ retrieves the resource from ACME, follows the link to the representation of dbpedia, and from there all following `owl:sameAs` relations. The deduction rules (1) and (2) ensure that the correct inferences with respect to symmetry and transitivity are derived from retrieved

`owl:sameAs` statements. The materialization of the transitive symmetric closure of `owl:sameAs` allows that all links are followed that represent the stadium via the request rule (2).

Additionally, if a triple with the property `foaf:depiction` is found, the schema of foaf is retrieved, where `foaf:depiction` is defined as the inverse property of `foaf:depicts`. Due to deduction rule (3) the linked program materialises inverse triples accordingly to the definition of the schema.

Instead of the three specific deduction rules, the linked program could also be built using e.g., the complete OWL LD rule set[c]. However, the defined deduction rules are sufficient for the task of finding pictures of the stadium.

Executing $\mathcal{P}$ returns 9 distinct results for Q, i.e., the program identifies 9 different pictures that can be displayed by the application[d]. The program follows links to 71 other representations of the stadium via `owl:sameAs`. The result graph $G_{\mathcal{P}}$ contains 8318 distinct triples. However, not all triples in the result graph represent original information about the stadium, e.g., a relation to the architect of the stadium can be given multiple times with properties from different vocabularies. The result graph contains 216 unique properties that are directly connected with the ACME representation of the stadium, with 213 properties derived from other resources.

Table 3 shows a comparison of the result graph from a program with custom rules as described in the example, with a program that uses no deduction rules or the complete OWL LD rule set.

Without deduction rules, the program only follows one link and identifies only one picture of the stadium. In contrast employing the complete OWL LD rule set results in the same amount of identified pictures, but materialises significantly more triples and properties, which are not necessary for the goal of the application.

[a] `http://dbpedia.org`; retrieved 1015-04-10.
[b] `http://xmlns.com/foaf/spec/`; retrieved 1015-04-10.
[c] `http://semanticweb.org/OWLLD/`; retrieved 2015-04-10.
[d] As ACME and its Web presence is fictitious, we simulate the initial resource described in the example, which contains a link to an actual LOD Web resource.

Table 3: Result sizes of programs with different complexity retrieving information about a sports stadium.

Deduction Rules	Retreived Pictures	Equivalent Resources	Direct Properties	Total Triples
none	1	1	3	3
custom	9	71	216	8318
OWL LD	9	71	2098	241745

When looking at different data-processing scenarios an important aspect is the speed at which the retrieved data changes. While data on the Web can generally be considered dynamic (i.e., changing over time), the previous examples can be seen as relatively stable. In contrast, data like stock trading information or sen-

sor measurements change multiple times per second. Applications that monitor such information can require to execute linked programs repeatedly with high frequencies. Even if the processed data does not change multiple times per second, the requirement to immediately react to a change can imply the repeated execution of queries.

✎ **Example 12**

ACME displays live updates of sport match scores on its homepage. To acquire the required information ACME aggregates news of sports events from various sources.

During a soccer match the following query might be executed with a program, that retrieves and data from other sports news providers:

```
SELECT ?home ?guest ?score
WHERE {
  ?x rdf:type ex:SoccerMatch.
  ?x dbpedia:hometeam ?home.
  ?x dbpedia:awayteam ?guest.
  ?x dbpedia:score ?score.
}
```

To be competitive ACME wants to display updates as soon as they are published. Therefore, the linked program is executed repeatedly as fast as possible potentially over data from several sources.[a]

[a] In this example we assume the sources do not limit the number of requests per minute, which might only be realistic if ACME pays fees to the data providers.

While in Example 11 the challenge lies in a speedy one-time processing of a large amount of data from distributed sources, in Example 12 the challenge is rather the repeated processing in fast succession of smaller or medium sized data sets.

There is an apparent implication, that smaller queries over smaller data sets can be processed faster and can therefore be executed repeatedly with higher frequencies. However, in this chapter we are also interested in the effects of our proposed execution model for parallel processing on the achievable frequencies. In particular, we study the impact of the overhead stemming from the management of several threads when executing a linked program multiple times.

3.3 PROCESSING ARCHITECTURE

In this section we describe an architecture to interpret linked programs, specifically we detail the query planning and summarise the evaluation of linked programs.

A Web-based application obtains required information by evaluating queries over an only implicitly defined dataset. The dataset consists of initial requests and triples, which contain links to further resources. A linked program defines the specifications of how to expand links and map vocabularies.

The purpose of the architecture is to find results for queries from a graph which is extended by the deductions and network lookups as defined in a linked program. The processing thus includes query and rule processing in tandem with the fetching of resources. The optimisation goals are to increase throughput, i.e., process as many triples and resources lookups as quickly as possible to decrease the runtime of the application. (i.e., return results as quickly as possible).

The architecture for accessing and processing Linked Data should exhibit the following characteristics:

- All algorithms should work in streaming fashion to reduce overall runtime by enabling results to be returned incrementally. Incremental processing is required, because the lookups are derived from the processed data, and lookups should be done as soon as possible to account for latency in network sources. Our assumption is that interleaving data and request processing reduces overall elapsed time relative to step-wise evaluation.

- The workload to process linked programs and queries should be as small as possible to not strain the system resources unnecessarily. To minimise the workload implies the applications of optimisations to reduce duplicates on data and requests. We want to reduce peak size of intermediary results (i.e., size of queues) during program evaluation, as well as time and space requirements. Afrati and Ulman [2] identify derivations as measure of runtime complexity. A reduction of duplicates implies a reduction of unnecessary derivations.

- The processing should be done in parallel processes to be able to leverage multi-core machines. The parallelisation should approach linear speed-up with respect to the employed cores. In particular, we have to balance I/O-bound and CPU-bound processes based on a data-driven execution model.

A physical operator plan is responsible for identifying result bindings for the BGPs in rule bodies and queries. The physical operator plan is derived from queries and rules in three steps:

1. Parse the program and the queries, and build an internal representation of rules and queries.

2. Create the logical operator plan from the rule bodies and BGP queries encoded with the equivalent of named relational algebra expressions. The logical operator plan contains cycles due to the recursive application of derivations from deduction rules. We generate the logical plan in a way that minimises the size of intermediate results.

3. Create the physical operator plan encoded with the equivalent of unnamed relational algebra expressions. The use of unnamed relational algebra implies that tuples instead of entire solution mappings are used in the physical plan. Omitting variables from the solution mappings saves memory and makes elimination of common subexpressions easier.

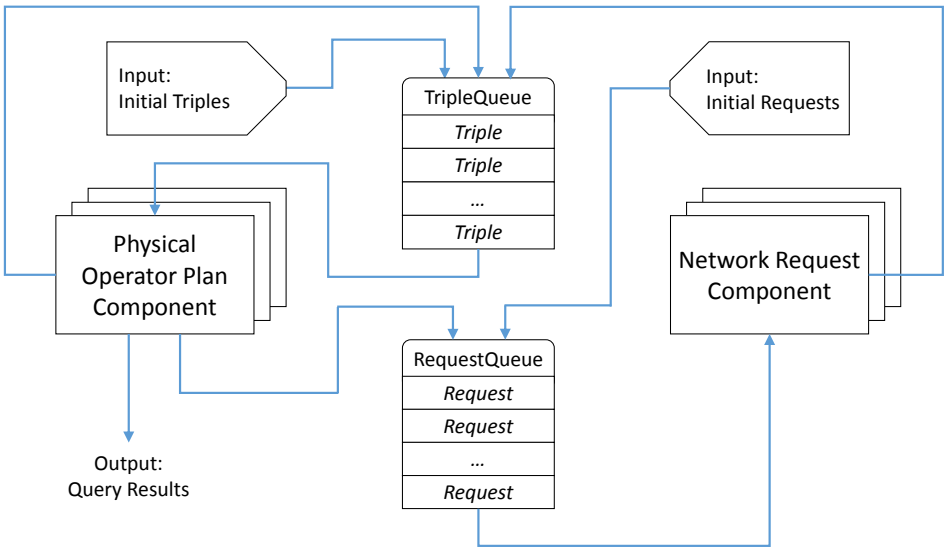


Figure 6: Illustration of the data flow between physical operator plan and request components of a system to process linked programs.

We describe details of the physical operator plan in Section 3.3.2.

The request component is responsible for carrying out HTTP requests. We describe details of the network request component in Section 3.3.1.

Figure 6 illustrates the dataflow between components of the system. Input to the system are initial triples and resources; output are query results. Data is input to and output from the system in a streaming fashion. Both physical operator plan and request components are multi-threaded.

We use concurrent sets for duplicate elimination (see Sections 3.4.4), and concurrent multimaps for storing the intermediate results of joins (see Section 3.3.2).

In the overall system we use two concurrent queues for passing work items between components, thus resolving the cycles between operators and components:

- TripleQueue q^t: for triples to be processed by the physical operator plan.
- RequestQueue q^r: for requests to be performed by the request component.

The RequestQueue is used to pass requests from the physical operator plan to the request component. The TripleQueue is used to pass triples from the request component to the physical operator plan and feed derived triples back into the physical operator plan.

We need the queues to break the cycle between the components, to avoid arbitrary deep call stacks due to the recursion. The cycle between physical operator plan and request component requires a queue as buffer due to different parallelisation and processing speeds. For the system to start, there has to be at least a triple in the TripleQueue or a request in the RequestQueue (see Definition 11).

Having constructed an optimised physical plan, the system evaluates the physical plan in parallel, and perform requests in parallel. Processing triples and carrying out requests is done with separate thread pools:

- TripleWorkers, which take triples from the TripleQueue for processing, and push derived triples back into the TripleQueue and derived requests into the RequestQueue. Let $W^t = \{w_1^t, ..., w_n^t\}$ be the set of triple workers, with n the number of workers.
- RequestWorkers, which take request objects from the RequestQueue, and push triples resulting from HTTP `GET` requests into the TripleQueue. Let $W^r = \{w_1^r, ..., w_m^r\}$ be the set of request workers, with m the number of workers.

The workers operate until the system determines that the processing should be finished. Conditions for the processing to finish include:

- a specified timeout has been reached;
- the system has requested a pre-defined number of resources;
- the system has reached a pre-defined depth in traversing the graph of resources, starting from the initial resources; or
- the computation has reached a fixpoint, i.e., the result graph is completely calculated according to the rules of the linked program and all queries are completely evaluated.

In the following we consider reaching the fixpoint as termination condition. The cycles between components complicate the detection of the fixpoint, especially when performing the processing in parallel. That the fixpoint is reached should be determined as quickly as possible to decrease the runtime of the system, which is non-trivial given the parallel setup with queues.

The next Sections 3.3.1 and 3.3.2 introduce the main components of the system, the physical operator plan and the request component, in more detail. The parallel execution of, and coordination between request component and physical operator plan is topic of Section 3.4.

3.3.1 *Network Request Component*

We assume data is distributed across HTTP-accessible resources, with links between these resources. The task of the request component is to carry out requests as specified in request rules. These requests are incrementally generated during program evaluation (link-following), based on the request rules.

The request component has the following requirements:

- Retrieve data as fast as possible, to not become the bottleneck in processing.
- Do not repeat equivalent requests.
- Carry out requests in a polite fashion, that is, do not overload servers with parallel requests.

Input to the request component are requests $(\mathsf{GET}, u, \emptyset)$ via the RequestQueue. The requests are represented by their URIs u as we allow only for lookups. Output are triples from parsing the payload D^{res} of the responses of the requests $\beth((\mathsf{GET}, u, \emptyset)) = (c, D^{res})$. If the response to a lookup is a redirection indicated by $c = 3xx$, the request component follows the redirect, i.e., performs another lookup on the referred resource. The output triples are fed into the TripleQueue so that the triples can be further processed.

To prevent multiple equivalent requests on the same resource, the request component can maintain a set of all URIs that were already used for requests in a visited set. Multiple requests on the same URI can then be detected and rejected by the request component. Multiple requests on the same URI might also occur if a previously unknown URI is requested that simply redirects to an already seen URI. Consequently, the system also needs to store information about redirects, i.e., which URIs redirected to the information resource that eventually delivered data. To prevent unnecessary requests rather than just filtering out the arriving duplicate triples is advantageous, to reduce source server load, network traffic and the amount of I/O-bound work.

To achieve a speedy but polite execution of a large number of requests across many Web servers, the request components can implement several best-practices for HTTP interactions, such as:

- Adherence to robots.txt[50] declarations.
- Implementation of a wait time between consecutive requests to the same host[51].
- Dead host detection, i.e., have requests to resources on servers that do not reply timeout. A long timeout threshold can lead to a prolonged program evaluation, if there is an unresponsive server. However, a too short timeout threshold might cause the system to not retrieve all available resources.

3.3.2 *Physical Operator Plan Component*

The physical operator component is used to identify the result bindings for the BGPs in rule bodies as well as queries:

- Result bindings of deduction rule bodies are used to generate derivations with the rule head.
- Result bindings of request rule bodies are used to generate requests with the rule head.
- Result bindings of queries are the output of a program.

The physical plan contains operators which actually carry out data processing operations to calculate the result bindings. The operators are connected

[50] http://www.robotstxt.org/; retrieved 2015-04-10.

[51] It used to be customary to wait several seconds between requests to the same server. In our implementation used for the experiments in Section 3.5 we just make sure that requests are made in sequence, that is, there are no parallel requests to the same host.

Table 4: Operators used in the logical and physical plans to process data.

Operator	Description
Input	Distinguished node for input of triples
TriplePattern	Evaluate triple patterns
EquiJoin	Compute equi-join between two inputs
Binding	Store intermediate results; only in physical plan
Project	Projection of tuples for `SELECT` queries
Output	Output query results
Derivation	Generate derived triples
Request	Generates requests

via sender/receiver relationships. The workers propagate solution sequences through the physical operator plan along these sender/receiver relationships.

During the generation of the logical operator plan, we use a heuristic to minimise the amount of intermediate results. We employ a version of the approach in [117], where the number of variables in a query is used as estimate for the cardinality of results. Thus, the logical plan is constructed in such a way that more selective joins are carried out first.

Both logical and physical operator plans consist of a finite set of operators (see Table 4). The logical and physical plan use the same operators, with the exception of the EquiJoin operator, which is extended in the physical plan by two Binding operators (see Section 3.3.3). The translation of the logical plan to the physical operator plan includes an optimisation to eliminate common subexpressions, i.e., equivalent operators are reused.

Rather than using named relational algebra to represent the solution sequences as in the logical plan, the physical plan does not know about variables, and therefore uses unnamed relational algebra expressions. We express the solution sequences as tuples of RDF terms, and do not need to carry the full variable-term mapping. The position of an RDF term in the tuple determines which variable maps to the term.

Individual triple patterns of BGPs of queries and rule bodies are represented with TriplePattern operators. BGPs with multiple triple patterns lead to multiple TriplePattern operators, connected via EquiJoin operators to represent join conditions (see Definition 7). Thus, each BGP is represented as a tree, and the combination of all BGPs in the program and queries form a forest of join trees[52].

The Input operator is the distinguished node in the operator plan, the place where to input data, which connects to all TriplePattern operators.

The trees receive tuples via the TriplePattern operators, which in turn receive triples from the Input operator. Depending on whether the BGP is from a query, a deduction rule or a request rule, the top operator of the BGP tree is connected

[52]Due to the reuse of equivalent Binding operators, BGPs are only trees in the logical operator plan. In the physical operator plan, the forest forms a connected graph, rather than a set of independent join trees.

to an Output operator (via Project operators), to a Derivation operator, or to a Request operator.

Instead of a directed acyclic graph of operators, which is the basis for most dataflow-systems, our operator graph may contain cycles. The possible cycles are established by connections from the Derivation operators to the Input operator, i.e, the cycles reflect the recursive application of deduction rules. That is, the Derivation operators send the generated triples back to the Input operator. We break cycles in the physical operator plan with the TripleQueue between Derivation operators and Input operator.

The RequestQueue decouples the physical operator plan and request component, to allow for taking into account the different processing speeds and different worker threads.

✎ **Example 13**

Listing 6 shows the rules of a linked program, which encodes that the property `foaf:depiction` is the inverse property of `foaf:depicts` with a deduction rule (1). Further a request rule (2) specifies that all resources that represent something, which is found to be a member of a team have to be retrieved. The rules of the linked program are to be executed together with the following query to retrieve pictures of teams:

```
SELECT ?y
WHERE { ?x foaf:depicts ?y .
        ?x rdf:type p:Team . }
```

The resulting logical operator plan is shown in Figure 7. Every triple pattern is represented with its own TriplePattern operator. Two EquiJoin operators represent the join variable `?x` in the query and the join variable `?n` in the request rule. The query results are projected to variable `?y`.

The physical operator plan derived from the logical plan is shown in Figure 8. The variable names are replaced, as only the position is relevant. The duplicate TriplePattern operators are removed, as common subexpressions have been eliminated. The EquiJoin operators are extend by two BindingOperators, where one BindingOperator is shared between both joins.

Listing 6: Linked program which encodes the inverse of foaf:depicts and the retrieval of resources representing members of teams.

```
@prefix p: <http://acme.example.org/vocabulary#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix http: <http://acme.example.org/api/> .
@prefix httpm: <http://acme.example.org/vocabulary#> .

# (1) Inverse of foaf depicts
{ ?u foaf:depicts ?v . } => { ?v foaf:depiction ?u . } .

# (2) Retrieve team members
{ ?n rdf:type p:Team
  ?m p:memberOf ?n . } => { [] http:mthd httpm:GET;
```

```
12 ||                                    http:requestURI ?m .} .
```

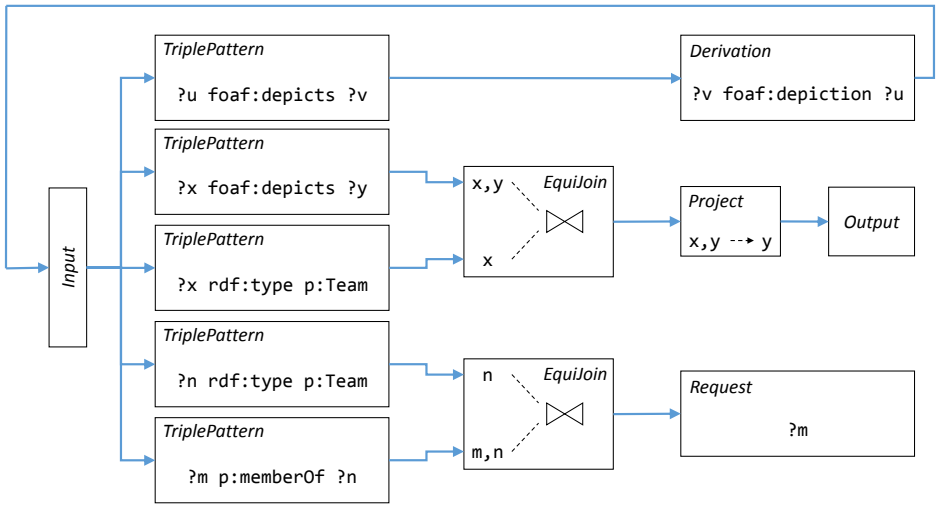


Figure 7: Logical operator plan for linked program in Listing 6 and query in Example 13.

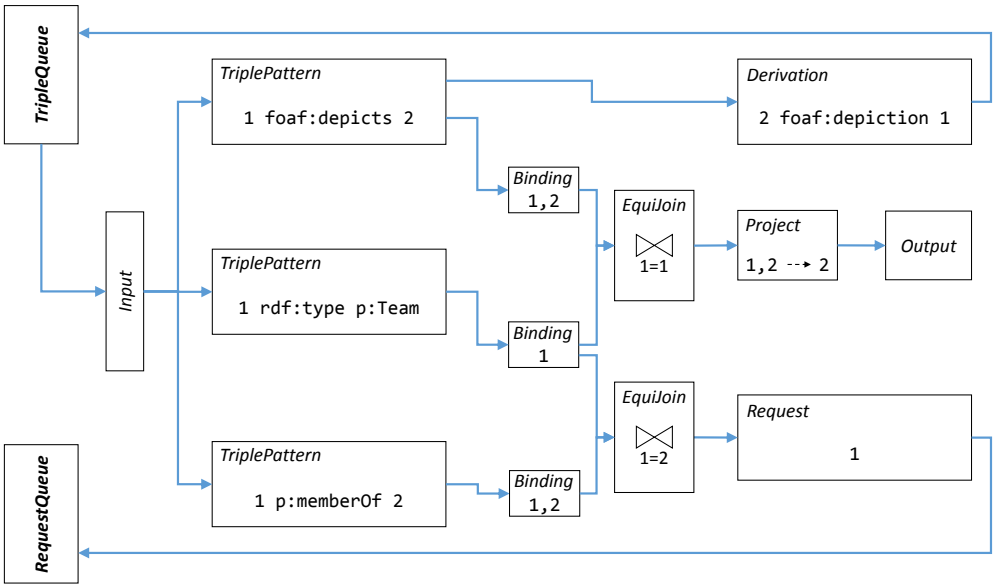


Figure 8: Physical operator plan for linked program in Listing 6 and query in Example 13.

An illustration of the physical operator plan for the complete OWL LD rule set can be found in Appendix A.1 in Figures 35 and 36.

With the exception of the EquiJoin operator (in conjunction with the Binding operators), none of the operators store intermediate results. Therefore, the workers can use the logic of the operators in multiple parallel processes without interfering with each other. In the following Section 3.3.3 we describe the algorithm upon which the EquiJoin operator is built, which allows to include the operator in the parallel evaluation setting of the system.

3.3.3 *EquiJoin Operator*

The EquiJoin operator takes as input tuples, and emits tuples according to the join condition. Our parallel algorithm to evaluate join operations in the physical plan is based on the symmetric hash join [128]. Consequently, we call our algorithm the parallel symmetric hash join.

Let $B = (b_1, ..., b_n)$ be a tuple of RDF terms that represents a solution mapping $\mu : \mathcal{V} \to B$.

Every EquiJoin in the logical plan is realised in the physical operator plan with two kinds of operators:

- A left and a right Binding operator, which store arriving tuples from preceding operators.
- An EquiJoin operator, which carries out the computation of the join and pushes the combined tuple to subsequent operators.

A physical EquiJoin operator receives tuples from its left and right Binding operator. The join operators contain information at what position in an arriving tuple the term is located to which the join variable is mapped. b_j denotes the solution in an arriving tuple for the join variable of the EquiJoin operator. The physical EquiJoin operators do not require any information about the names of the variables to which the RDF terms in the propagated tuples are mapped. The information about variable names is only required in the Output operators, which construct the query results.

The Binding operators that feed an EquiJoin operator store the arriving tuples from their preceeding operators in the plan in multimaps. The multimaps relate solutions to sets of tuples that contain the solution at join position, similar to traditional symmetric hash joins. The key of an entry in a multimap is the solution b_j for the join variable of the join operator in the arriving tuple. The values of the map are the subsets of all arrived tuples, which have the key value b_j is in join position.

Let $M^x = \{b_j \to \{B | b_j \in B\}\}$ be the left and right multimap of an EquiJoin with $x \in \{l, r\}$ and b_j the term in join position.

Unlike operator plans with traditional symmetric hash joins, the approach to differentiate between the Binding and EquiJoin operators allows us to reduce memory consumption during evaluation. We can use the same physical Binding operator for several EquiJoin operators if the EquiJoin operators evaluate the same triple pattern on the left or right side, i.e., elimination of common subexpressions in the plan.

A multimap of a Binding operator provides two basic operations: put tuples into the map and read sets of tuples from the map:

- $M^x.put(A)$: adds the tuple A with $b_j \in A$ to the set in value position of the entry $(b_j \to \{B | b_j \in B\})$ in M^x. Note the entry for the key b_j is created, if the entry does not exist yet in the map.

- $M^x.read(b_j)$ returns the set in value position of the entry $(b_j \rightarrow \{B | b_j \in B\})$ in M^x, i.e., the set of all tuple with b_j in join position.

Both operations are non-blocking, i.e., multiple threads can read and write in parallel. A read operation only returns the tuple for which the write operation has been completely finished.

For brevity we define

$$\text{Let } \bar{x} = \begin{cases} l & \text{if } x == r \\ r & \text{if } x == l \end{cases}$$

When a tuple B^x is propagated to a join node from its left (right) predecessor, the join node produces combined tuples by joining B^x with the matching tuples $M^{\bar{x}}.read(b_j)$ in the right (left) map. Algorithm 1 describes the parallel symmetric hash join as implemented in the join operator, which allows parallel access by multiple threads.

Algorithm 1: Parallel Symmetric Hash Join.

Input : $\Upsilon^x = \{B\}$ with $x \in \{l, r\}$ incoming tuples from left and right operator
Output: Σ outgoing joined tuples

```
1 foreach B^x ∈ Υ^x parallel do
2     Σ ← ∅
3     M^x.put(B^x)
4     Θ ← M^x̄.read(b_j^x)
5     forall T ∈ Θ do
6         add (B^x ∪ (T \ {b_j^x̄})) to Σ
7     Push Σ
```

We consider sets $\Upsilon^x = \{B\}$ with $x \in \{l, r\}$ of incoming tuples from the left or right predecessor. The sets Υ^x are constantly filled with tuples from preceding operators. In particular multiple tuples can be pushed to the same Binding operator by several threads simultaneously. Every tuple in one of the input sets is processed in parallel (line 1). Σ will contain the joined tuples for an incoming tuple and is initialized as empty set (line 2).

In line 3 the incoming tuple B^x is added to the corresponding map, where $x \in \{l, r\}$ indicates if the tuple was pushed to the left or right Binding operator.

Next, the set of tuples from the opposite map $M^{\bar{x}}.read(b_j^x)$ is retrieved, where RDF term $b_j^{\bar{x}}$ in join position matches RDF term b_j^x in join position of B^x (line 4).

Finally, in line 6 every retrieved matching tuple from the opposite hash map is combined with B^x and stored in Σ. We remove the RDF term in join position from the retrieved tuples, as that term is also present in B^x.

The EquiJoin operator pushes the tuples in Σ to subsequent operators within the same thread, in which the incoming tuple B^x was pushed to the EquiJoin operator.

Next, we show that the join algorithm is correct in a parallel scenario, i.e., all joined binding lists are generated for in parallel arriving binding lists.

↣ Lemma 1

$\forall B^l \in \Upsilon^l$ and $\forall B^r \in \Upsilon^r$ eventually $B^l \cup B^r \in \Sigma$ is generated,
iff $b_j^l \in L = b_j^r \in R$

Multiple TripleWorker threads act in parallel on the same multimaps M^r and M^l in the Binding operators connected to an EquiJoin operator. Every TripleWorker thread w_i^t accesses the maps at only two points during processing of a binding list B^x at a join node:

- Write access on the multimap M^x in line 3 to create an entry in the hash map for B^x. We denote $\mathcal{W}_B^x$ for this access operation.
- Read access on the opposite multimap $M^{\bar{x}}$ in line 4 to retrieve the matching binding lists for B^x. We denote $\mathcal{R}_B^{\bar{x}}$ for this access operation.

We further denote $\phi \ll \psi$, if an operation ϕ is completed before an operation ψ starts.

Proof Sketch

When a TripleWorker w_i^t pushes a tuple B^x to an EquiJoin, the TripleWorker first creates an entry in M^x, before the matching tuples from $M^{\bar{x}}$ are retrieved. Because both operations are done by the same thread, it holds trivially true that

$$\mathcal{W}_B^x \ll \mathcal{R}_B^{\bar{x}} \tag{1}$$

We assume that there is a situation when we do not create a joined tuple for two arriving and matching tuples:

Assumption:

$$\begin{aligned} &\exists U \in \Upsilon^l \text{ and } \exists V \in \Upsilon^r \text{ with } b_j^l \in U = b_j^r \in V \\ &\qquad \text{and } (U \cup V) \in \Sigma \text{ is not generated.} \end{aligned} \tag{2}$$

Eventually, a TripleWorker creates for every tuple $B^x \in \Upsilon^x$, which he pushes to an EquiJoin an entry in the corresponding map M^x (line 3). During the processing of a tuple B^x, we generate all joined tuples in the loop in lines 5 to 6 with the matching binding lists retrieved from $M^{\bar{x}}$.

Let w_i^e be the TripleWorker thread that pushes U.
Let w_j^e be the TripleWorker thread that pushes V.

If w_i^e cannot generate the joined tuple $(U \cup V)$, it follows that w_i^e reads M^r before the entry for V in M^r is written by w_j^e. If w_j^e cannot generate the

joined tuple $(U \cup V)$ either, it follows analogously that w_j^e reads M^l before the entry for U in M^l is written by w_i^e:

$$\mathcal{R}_U^r \ll \mathcal{W}_V^r \text{ and } \mathcal{R}_V^l \ll \mathcal{W}_U^l \tag{3}$$

A simultaneous read and write access on the same map M^x is also covered in (3), as the write operation has to be completely finished, before the written entry is returned by the read operation.

If (3) holds true it is a contradiction to (1):

$$\text{(3) with (1): } \mathcal{W}_U^l \ll \mathcal{R}_U^r \ll \mathcal{W}_V^r \ll \mathcal{R}_V^l \text{ and } \mathcal{R}_V^l \ll \mathcal{W}_U^l \tag{4}$$

$$\implies \mathcal{W}_U^l \ll \mathcal{R}_V^l \text{ and } \mathcal{R}_V^l \ll \mathcal{W}_U^l \quad \text{contradiction} \tag{5}$$

Therefore the assumption in (2) can not hold true and we have shown Lemma 1.

□

Intuitively we do not miss a join result in the parallel processing of two tuples U and V, because if the processing of U does not generate the result, the processing of V will generate the result and vice versa.

However, the parallel processing of two arriving triples can lead to the duplicate generation of a joined tuple in case it holds true that

$\mathcal{W}_U^l \ll \mathcal{R}_V^l$ and $\mathcal{W}_V^r \ll \mathcal{R}_U^r$.

If both, the entry for U is written to M^l and the entry for V is written to M^r before the respective TripleWorker threads read M^l and M^r, the resulting joined binding tuple is generated twice (see Section 3.4.4).

3.4 COORDINATION OF DATA PROCESSING AND NETWORK REQUESTS

In the following we first describe different threading models to implement a push-based execution model for the parallel evaluation of linked programs (Section 3.4.1). In particular, we show how different threads for evaluating the physical operator plan and performing network requests can be coordinated in a threading model that allows to actively track when the termination condition of the processing is reached (Section 3.4.2). Further, we describe how the number of active TripleWorker and RequestWorker threads can be dynamically determined during runtime to improve hardware utilisation (Section 3.4.3). Finally, we describe how duplicate solutions can arise during processing and such duplicates can be handled(Section 3.4.4).

3.4.1 *Threading Models*

For parallel evaluation of programs [62] propose an execution model, where every operator iterates over the data items it has produced and pushes these results to other processes (i.e., threads), which execute subsequent operators in the evaluation plan. Thus, every operator is executed in its own process, which allows for the parallel evaluation of query plans.

In contrast we propose an execution model that does not use an individual thread for every operator in the physical operator plan. Instead, we allow operators to schedule each other within a single thread. Therefore we avoid the overhead of the thread scheduling by the operating system as well as overhead of inter-process communication [41]. In particular, our approach allows to determine the number of employed parallel processes freely, as the number is not implied by the amount of operators in the plan.

Traditional execution models [41, 48], which allow operators to schedule each other in the same processing thread, are demand-driven. In demand-driven execution models operators request data items from preceding operators as required for the processing. The operators iterate over the received (pulled) data items to produce new (intermediate) results. In contrast, we use a data-driven push-based scheduling of operators, where the operators immediately push intermediate results to subsequent operators within the same thread along the sender-reciever link in the physical operator plan.

Such a push-based execution model especially caters to data processing scenarios that include network lookups, as data can be processed immediately when it arrives, rather than waiting on operators during slow network request. The advantage of a push-based model in scenarios where data-sources have to unload data as it arrives, is also pointed out in [41].

A worker thread can be in one of three states:

IDLE Worker tries to acquire a new element from the queue.

PROCESSING Worker processes an element from the queue.

SLEEPING Worker is set to sleep when its queue is empty.

We consider a linked program $\mathcal{P}$ to be completely processed, when the result graph $G_{\mathcal{P}}$ is materialised and all result bindings of the registered queries are found (i.e., its fixpoint is reached). The decision when a linked program is completely evaluated and the processing can be stopped has to be made at runtime as the data that is processed is not completely known before the processing starts.

During processing the fixpoint of a linked program is reached, once two termination conditions are met:

- All queues are empty, i.e., there are no further triples or requests to process.
- All worker threads are idle or sleeping, i.e., the worker threads are not processing triples or requests.

To terminate the processing simply when the queues are empty is not sufficient, as workers could be currently processing an item (i.e., triple or request) and add results to the queues again.

In the following we describe different threading models that can be used to realise a push-based execution model including an approach to check for the termination conditions.

SERIAL There is only one thread, with a dual role as TripleWorker and RequestWorker. The thread takes one item (triple or request) from one of the queues and processes the item. Derived triples and requests are stored in the respective queues. There is no parallel processing, and all operations are carried out sequentially. The check for the termination condition is straightforward: When the thread tries to take a new item (i.e., is idle), but both queues are empty, the fixpoint is reached. We use the single-threaded serial model as baseline.

ROUNDS A naive approach to a multi-threaded model is to use dedicated TripleWorker and RequestWorker, which work in conjunction to process a linked program in rounds: Triples and requests for the current round n are taken from the respective queue and processed by the workers. In round n derived triples and requests are stored into new queues for the next round $n+1$. Once the queues in round n are empty, round $n+1$ starts. Every round corresponds to an execution step of all rules $step(\rho, G) \forall \rho \in (P^d \cup P^r)$, where G are the triples in the TripleQueue. The rounds model does not fully leverage the available system resources towards the end of each round: As a queue contains fewer items than there are threads, only some of the threads are processing the last items while the other threads are sleeping, even though the queues for round $n+1$ might already contain items that could already be processed by the sleeping threads. The rounds approach allows for an easy identification of the fixpoint, which is reached once the queues are empty at the beginning of a new round. At the beginning of a round all threads are necessarily idle.

SPINNING Several dedicated TripleWorker and RequestWorker take and process items from the respective TripleQueue and RequestQueue. Derived triples and requests are directly pushed back into the same queues. If a worker is idle but finds its queue to be empty, the worker sleeps for a time and periodically wakes up to try to acquire an item from the queue again (i.e., busy waiting). The wait time can be freely chosen and results in a trade off: A short waiting time can cause system overhead as workers often wake up unnecessarily; a long waiting time can cause the workers to stay unnecessarily asleep while items in the queues are present. To identify the fixpoint the main thread of control checks periodically whether i) the queues are empty and ii) all workers are in sleeping state.

BLOCKING Several dedicated TripleWorker and RequestWorker take and process items from the respective TripleQueue and RequestQueue. Derived triples and requests are directly pushed back into the same queues. If a worker tries to take an item from an empty queue, the queue blocks the worker,

i.e., puts the worker to sleep. If a new item arrives at the TripleQueue or RequestQueue, the queue signals all sleeping TripleWorker and RequestWorker respectively, and the workers wake up. The main thread of control has program logic that actively tracks how many workers are sleeping and if the queues are empty to identify the fixpoint (see Section 3.4.2). As the blocking approach directly controls the state of the worker, it might leverage the system resources better. However, the necessary program logic to determine the fixpoint can also cause more overhead compared to *rounds* and *spinning*.

Apart from the single-threaded *serial* baseline approach, we focus on architectures with dedicated TripleWorkers and RequestWorkers, i.e., the threads do not change their role from executing I/O-bound tasks to core-bound tasks or vice-versa. We reserve the development of models where threads can change their role for future research work.

3.4.2 *Blocking Worker Control*

Although there are queue data structures that allow for parallel access to some extent[53], a single TripleQueue and RequestQueue can still be a bottleneck for the execution model, due to lock contention during some phases of the processing. Lock contention refers to a situation in which multiple threads try to access the same data object simultaneously; all but one thread have to wait as the object can only be accessed by one thread at a time.

We extend the blocking approach by employing multiple TripleQueues and RequestQueues, so that adding and reading from the queues can be done in parallel without interference. In particular we use an individual queue for every worker. While employing multiple queues can reduce lock contention it requires additional overhead for coordination, e.g., to evenly balance the workload for the workers the queues should be filled evenly.

We now describe the *blocking* model with multiple queues in more detail. A simplification of *blocking* with only single queues is straight-forward.

Every TripleWorker (RequestWorker) only takes triples (requests) from its associated queue.

Let $Q^t = \{q_i^t | \exists q_i^t \forall w_i^t \in W^t\}$ be the set of TripleQueues of the TripleWorker threads. Let $Q^r = \{q_i^r | \exists q_i^r \forall w_i^r \in W^r\}$ be the set of RequestQueues of the RequestWorker threads.

Every queue q_i^x with $x \in \{t, r\}$ has an associated lock, which a worker must hold to perform read and write operations on the queue. All workers act in parallel, but not more than one worker can hold a specific lock simultaneously. If a worker tries to acquire a lock that is already taken, the worker waits until the lock becomes free again, i.e., lock contention. However, the use of an individual queue for every worker reduces the incidence of lock contention.

[53] e.g., ConcurrentLinkedQueue in Java: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html; retrieved 2015-04-10.

Derived triples and triples from requests are pushed into a randomly chosen TripleQueue. Derived requests are pushed into a randomly chosen RequestQueue. The random choice of the queues is done with an equal distribution and results in an approximately even dispersion of the overall workload across workers.

While read and write operations on an individual queue have to be synchronised with the associated locks, the actual processing after taking an item from a queue does not require to keep the queue locked for a write operation. E.g., if an TripleWorker takes a triple from its TripleQueue, the queue is locked for write operations from other worker threads. However, when the TripleWorker starts to process the triple, the lock of the TripleQueue can be released immediately for other workers to add items.

We use a boolean vector to store whether the termination conditions are met. Every entry in the vector corresponds to a worker.

Let $V = (v_1, ..., v_k)$ with $v_i \in \{true, false\}$ and $k = n + m$ be a boolean vector, where n is the number of TripleWorker and m the number of RequestWorker.

An entry in the vector is set to false if the corresponding worker is set to sleep, as the associated queue of the worker is empty. Therefore, an entry summarises the state of a worker and its queue. The use of a boolean vector allows to check the termination conditions without interfering with processing workers.

Algorithm 2 describes how the workers are controlled and the fixpoint is determined during runtime. In the beginning we assume every worker is idle and at least one request or triple has to be in one of the queues. We continuously process until we reach the fixpoint (line 2). Every idle worker acquires the lock for its queue (lines 3 - 4). There are two main parts in the algorithm, depending on the state of the queue of the worker:

- If the queue of a worker is empty it is taken as an indicator that processing might have finished, and the process starts to test if a fixpoint has been reached (lines 5 - 16).
- If there is at least one element in the queue of a worker, the processing of the element starts (lines 17 - 26).

If the queue of a worker is empty, the vector entry of the worker is set to false (line 6). Next the locks of all queues are acquired and it is checked if all entries in the vector are set to false. If all entries are false, the fixpoint is reached and processing stops, as all worker threads are idle or sleeping and all queues are empty (lines 8 - 12). If at least one of the entries in the array is true, processing is ongoing and the worker is just set to sleep (lines 13 - 16).

If the queue of a worker is not empty, the worker takes an element from its queue, releases the lock of the queue and starts processing the element (lines 17 - 20). If the worker derives a triple or a request during processing, the derived element is pushed into a randomly chosen queue and the corresponding vector entries are set to true (lines 21 - 25). Finally, the worker thread of the chosen

queue is woken up (line 26); waking up an idle or processing worker has no effect.

Algorithm 2: Control of parallel workers threads

Input : W^x, Q^x with $x \in \{t, r\}$; the workers and queues
Output : *done*; is true when fixpoint is reached
Requirement: All triple workers $w_i^t \in W^t$ are idle
Requirement: All request workers $w_i^r \in W^r$ are idle
Requirement: At least one $q_i^x \in Q^x$ contains an element

```
1  done ← false
2  while ¬done do
3      foreach idle w_i^x ∈ W^x parallel do
4          AcquireLock( q_i^x )
5          if q_i^x is empty then
6              v_i ← false
7              forall q^x ∈ Q^x do
8                  AcquireLock( q^x )
9              if every v ∈ V == false then
10                 forall q^x ∈ Q^x do
11                     ReleaseLock( q^x )
12                 done ← true
13             else
14                 w_i^x.sleep()
15                 forall q^x ∈ Q^x do
16                     ReleaseLock( q^x )
17         else
18             e ← q_i^x.take()
19             ReleaseLock( q_i^x )
20             w_i^x.process(e)
21             AcquireLock( q_u^t and q_v^r ) with random u, v
22             v_u, v_v ← true
23             add derived triples of w_i^x to q_u^t
24             add derived requests of w_i^x to q_v^r
25             ReleaseLock( q_u^t and q_v^r )
26             wake up w_u^x, w_v^x
```

Next, we show that in Algorithm 2 we determine correctly that when a fixpoint is reached, i.e., *done* will be set to true, iff all queues are empty and all workers have finished processing. We have to show both directions:

1. When *done* is set to true, the fixpoint is reached; i.e., the evaluation of a linked program does not terminate to early (Lemma 2).

2. When the fixpoint is reached, *done* is set to true; i.e., if there is a fixpoint we do not miss it and the evaluation of a linked program terminates (Lemma 3).

↬ Lemma 2

$(done = true) \Rightarrow$
$(\ (\forall q_x^i \in Q^x :\ q_x^i = \emptyset)$ and $(\forall w_i^x \in W^x :\ w_i^x$ idle or sleeping$)\ )$

Intuitively: If $done = true$, all queues are empty and all workers have finished processing.

Proof Sketch

First we show that the assertion holds true that if an entry in vector v_i is $false$, the corresponding worker w_i^x is not processing.

$$v_i = \text{false} \implies w_i^x \text{ is idle or sleeping} \tag{6}$$

If an $v_i \in V$ is set to false in line 6, the corresponding worker w_i^x has to be idle, as asserted with line 3. Afterwards either the fixpoint is reached (lines 9 to 12) and no further processing will take place, or w_i^x is set to sleep (line 14). If any worker w_i^x is asleep it can only be awoken in lines 26, after v_i was set to true in line 22. Therefore, (6) holds true.

Further we show that the assertion holds true that if an entry in vector v_i is $false$, the corresponding queue q_i^x is empty.

$$v_i = \text{false} \implies q_i^x = \emptyset \tag{7}$$

Any $v_i \in V$ can only be set to false in line 6, if the corresponding queue q_i^x is empty (line 5). Elements can only be added to a q_i^x in lines 23 and 24. Before elements are added to q_i^x the corresponding v_i is set to true in line 22. Therefore, (7) holds true.

To show that we determine correctly that the fixpoint is reached, we assume that we determine the fixpoint to early.

Assumption:

$$\begin{aligned} &(done = \text{ true }) \text{ and} \\ &(\ (\exists q_x^i \in Q^x \text{ with } q_x^i \neq \emptyset) \text{ or } (\exists w_i^x \in W^x \text{ with } w_i^x \text{ processing})\ ) \end{aligned} \tag{8}$$

Intuitively, we assume that we determine that a fixpoint is reached, but either one of the queues still contains an element or one of the worker threads is still processing. Line 9 asserts that $done$ can only be set to true, if all $v_i \in V$ are false. However, this is a contradiction with (6) and (7) :

$$\begin{aligned} \implies &(\ (\forall v_i \in V\text{: } v_i = \text{ false }) \text{ and } (\exists q_x^i \in Q^x \text{ with } q_x^i \neq \emptyset)\ ) \text{ or} \\ &(\ (\forall v_i \in V\text{: } v_i = \text{ false }) \text{ and } (\exists w_i^x \in W^x \text{ with } w_i^x \text{ processing})\ ) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{contradiction with (6) and (7)} \end{aligned} \tag{9}$$

Therefore (8) can not hold true and we have shown Lemma 2.

□

↣ Lemma 3

$$\big(\ (\forall q_x^i \in Q^x :\ q_x^i = \emptyset) \text{ and } (\forall w_i^x \in W^x :\ w_i^x \text{ idle or sleeping})\ \big) \Rightarrow \text{ eventually } (done = true)$$

Intuitively: If all queues are empty and all workers have finished processing, $done$ will eventually be set to true.

Proof Sketch

We first show that the assertion holds true, that if a worker is sleeping and never been woken up again, its corresponding entry in V is false.

$$w_i^x \text{ forever sleeping} \implies v_i = false \tag{10}$$

A worker w_i^x can only be set to sleep in line 14, after its corresponding vector entry v_i is set to false. A vector entry v_i can only be set to true in line 22. However if a vector v_i entry is set to true, its corresponding worker will eventually be woken up in line 26. Therefore (10) holds true.

We assume that a the fixpoint is reached, but it is not identified:

Assumption:

$$\big(\ (\forall q_x^i \in Q^x :\ q_x^i = \emptyset) \text{ and } (\forall w_i^x \in W^x :\ w_i^x \text{ idle or sleeping})\ \big) \text{ and } (done = \text{ false }) \tag{11}$$

As all queues are empty and no processing takes place, it is trivial to see, that sleeping workers will not be woken up again.

Furthermore, every idle worker (line 3) with an empty queue (line 5) is either put to sleep (line 14) or $done$ is set to true (line 12). Therefore, we can simplify (11) to:

$$\big(\ (\forall q_x^i \in Q^x :\ q_x^i = \emptyset) \text{ and } (\forall w_i^x \in W^x :\ w_i^x \text{ sleeping forever})\ \big) \text{ and } (done = \text{ false }) \tag{12}$$

A worker can only be set to sleep in line 14. Setting workers to sleep can not be done in parallel as in line 8 the locks for all queues are acquired, which are released in line 16.

It follows that if $(\forall w_i^x \in W^x :\ w_i^x$ sleeping forever $)$ in (12) holds true (i.e., all workers are forever sleeping), there was a point during processing where all, but one worker w_k^x were forever sleeping. Line 6 asserts, that v_k is false when w_k^x is set to sleep. As $done = false$ in (12) it follows with lines 9 to 12,

that there has to be another v_l that is true, otherwise the fixpoint would have been identified. This is a contradiction to (10):

$$\implies \exists v_l \in V \text{ with } k \neq l \text{ and } v_l = \text{true and } w_l^x \text{ forever sleeping} \quad \text{contradiction with (10)} \tag{13}$$

Therefore (11) can not hold true and we have shown Lemma 2.

□

3.4.3 *Adaptive Processing*

The set of TripleWorkers W^t and the set of RequestWorkers W^r perform the evaluation of a program in parallel. The workers operate on the shared physical operator plan (including multimaps), which allows multiple threads to operate on the data in parallel. It can be freely configured how many workers should be there for data processing, and how many workers for request processing.

TripleWorker threads can be considered to be CPU-bound, i.e., in general a running TripleWorker thread utilises almost completely the core on which the thread is executed. Therefore the number of TripleWorker is driven by the number of available cores: Fewer threads than available cores would not fully utilise the capacity of the system and more threads can not generate a benefit, as the existing threads are already able to fully utilise the system [87].

RequestWorker threads can be considered I/O-bound, i.e., a running RequestWorker thread utilises only a fraction of the core on which the thread is executed, as requests are constrained by network bandwidth and latency. The number of RequestWorker threads can therefore exceed the number of available cores until the overhead of coordinating the threads outweighs the benefit of using parallel requests [87].

Operating systems rotate the threads that are active on the cores of a system. A too large amount of RequestWorkers threads can consequently also interfere with the efficiency of the TripleWorkers, as the the operating system can allot less processing time for the CPU-bound threads.

As the number of RequestWorker threads depends on network properties we introduce Algorithm 3, which adapts dynamically the number of employed RequestWorker to the average response times and amount of requests to be executed.

Thus, we improve the performance of the system for different execution scenarios with varying constraints in terms of requests.

Algorithm 3: Adaptive employment of RequestWorker threads

Input : T; threshold
Output : m; number of request worker threads during runtime

1 $m \leftarrow$System.availableCores()
2 **forall** i *in* $1..m$ **do**
3 w_i^r.open()
4 $reqCount_i \leftarrow 0$
5 $avgTime_i \leftarrow 0$
6 add w_i^r to W^r
7 **foreach** $w_i^r \in W^r$ ***parallel*** **do**
8 w_i^r.process($e \in q_i^r$)
9 $time_i \leftarrow w_i^r$.getResponseTime()
10 $avgTime_i = \frac{(avgTime_i * reqCount_i) + time_i}{(reqCount_i + 1)}$
11 $reqCount_i = reqCount_i + 1$
12 **if** $(avgTime_i * q_i^r.size()) > T$ *and* w_i^r.isOpen() **then**
13 w_i^r.close()
14 $m \leftarrow m + 1$
15 w_m^r.open()
16 $reqCount_m \leftarrow 0$
17 $avgTime_m \leftarrow 0$
18 add w_m^r to W^r
19 **if** $(avgTime_i * q_i^r.size()) < T$ ***and*** w_i^r.isClosed() **then**
20 w_i^r.open()
21 w_m^r.terminateWhenDone()
22 remove w_m^r from W^r
23 $m = m - 1$

Algorithm 3 takes threshold parameter T as input and determines during runtime the number of employed request worker threads m. We assume the *blocking* threading model with multiple queues.

A RequestWorker can be

OPEN Worker accepts new requests in its request queue.

CLOSED Worker does not accept new requests in its request queue, but keeps processing the requests that are already in the queue.

In lines 2 to 6 we initialise the first m RequestWorker, where m equals the number of cores available on the system. We set all RequestWorker to be open (line 3) and initialise two variables to count the performed requests (line 4) and to track the average response times (line 5).

All RequestWorker process in parallel requests from their RequestQueue (lines 7 and 8). When a RequestWorker is finished processing a request from its queue, we update the tracked average response time (lines 9 to 10) and increase the request counter (line 11).

We close a RequestWorker, if the currently expected time for the worker to process all requests in its queue (i.e., the average response time multiplied by the current queue size of a worker) exceeds the threshold T of an open worker (line 12). Further, we increase m (line 14) and open a new worker (lines 15 to 18).

We re-open a previously closed worker, if its average response time is again below the threshold T (lines 19 to 20). Further, we decrease m and signal one of the RequestWorker that was created when another worker was closed to terminate when it has finished processing the items in its queue (lines 21 to 23). The algorithm ensures that the initiated workers in line 3 will never be terminated, as for every closed worker, a new one is created.

The threshold parameter T can be set before execution depending on the employed hardware, where a PC with better parallelisation capabilities (i.e., more cores) implies a lower value for T. Thus, the decision on the number of employed RequestWorker is tied to local hardware capabilities, rather than a priori unknown and changing network properties.

The employment of Algorithm 3 allows the system to dynamically scale out the number of RequestWorker threads, when it is determined during runtime that there are a large amount of requests with higher response times to be done. However, as the system adapts over time it can not be expected that the system always acts with a strictly optimal amount of RequestWorker during processing. Consequently, a fixed amount of RequestWorker threads can outperform a dynamic scaling of threads; especially if the network properties are known and stable, the amount of RequestWorker can be chosen before runtime, so the system can continuously run with a close-to-optimal performance.

3.4.4 *Handling Duplicates*

During processing, the same tuple can be generated multiple times. Such duplicate solution mappings result in unnecessary derivations, and potentially increase the processing time. [49, 119] describe how duplicates can affect query evaluation time.

Such duplicates can be singular occurrences or the result of execution loops. Singularly occurring duplicates do not prevent the program evaluation from terminating. The same tuple can be generated in a singular instances due to the following cases:

- During an calculation of joins the same tuple be may produced twice, due to the parallel access to the multimap in a EquiJoin operator (see Section 3.3.3).
- Different data sources may contain the same triple, which might be retrieved causing the TriplePattern operators to produce the same tuples multiple times.
- Multiple rules can derive the same tuple independently of each other, as the same tuple is generated by disjoint paths in the operator plan. Similarly duplicate derivation paths can also result in the same request to be

executed more than once, which results the same (duplicate) triples to be retrieved.

> ✎ **Example 14**
>
> Listing 7 shows a linked program with two rules, specifying that every soccer player is also an athlete (1) and everybody who is member of a sport team is an athlete. The starting graph contains triples about somebody who is a soccer player and member of a team. The starting graph allows for the derivation that the described person is an athlete from both rules, i.e., two derivation paths in the operator plan lead to the same inferred tuple.

Listing 7: Linked program with two derivation paths leading to the same tuple.

```
# Starting graph
acme:johnDoe rdf:type p:SoccerPlayer ;
             p:memberOf acme:practiceTeam .
acme:practiceTeam rdf:type p:Team .

# (1) Every soccer player is an athlete
{ ?x rdf:type p:SoccerPlayer . } => { ?x rdf:type p:Athlete . } .

# (2) Every member of a sport team is an athlete
{ ?n rdf:type p:Team
  ?m p:memberOf ?n . } => { ?m rdf:type p:Athlete . } .
```

In contrast to the singular occurring duplicates, execution loops prevent the processing from finishing. An execution loop can occur as the result of the application of a cyclic operator plan and prevent that the system reaches a fixpoint. Instead, during the execution of the linked program the same tuples are repeatedly derived.

➳ Definition 12: Execution Loop

Let $\mathcal{P} = (G, R, P^d, P^r)$ be a linked program.
Let f and $step$ be the functions that map any rule and a graph to the derived graph and resulting graph respectively from one execution step of the rule (see Definitions 9 and 10):

$$f(\rho, G) = \begin{cases} f^d(\rho, G) & \text{if } \rho \in P^d \text{ a deduction rule} \\ f^r(\rho, G) & \text{if } \rho \in P^r \text{ a request rule} \end{cases}$$

$$step(\rho, G) = \begin{cases} step^d(\rho, G) & \text{if } \rho \in P^d \text{ a deduction rule} \\ step^r(\rho, G) & \text{if } \rho \in P^r \text{ a request rule} \end{cases}$$

Further, let F and $Step$ be the functions that map sets of rules P and a graph, to the derived graph resulting from one execution step of all rules in the set:

$$F(P, G) = \dot{\bigcup}_{\forall \rho \in P} f(\rho, G)$$

$$Step(P, G) = \dot{\bigcup}_{\forall \rho \in P} step(\rho, G)$$

There is an execution loop in program $\mathcal{P}$ if there is a subset of the result graph of the program $G \subseteq G_{\mathcal{P}}$ and a subset of rules $P \subseteq (P^d \cup P^r)$ so that a recursive execution of P over G eventually extends the graph G with the identical graph G:

$$F(P, (Step_1 \circ ... \circ Step_n)(P, G)) = G_{new} \quad \wedge G \subseteq G_{new}$$

Intuitively an execution loop occurs, if a series of recursive applications of a set of rules derives a graph, which triggers the same series of recursive rule applications.

Generally an execution loop of a linked program cannot be a priori predicted, as the data retrieved by request rules is not known before the program is executed. However, if a program $\mathcal{P} = (G, R, P^d, \emptyset)$ contains no request rules an execution loop can appear, if there is a subset of deduction rules $P^d_{loop} \subseteq P^d$, where the union of all rule bodies is a subgraph pattern (see Section 5.3 Definition 20[54]) of the union of all rule heads.

$$\forall \rho^d \in P^d_{loop} : \bigcup B \subseteq \bigcup H \text{ with } \rho^d : \{B\} \implies \{H\}$$

For the execution loop to appear all rules P^d_{loop} have to be triggered during execution of the program:

$$\forall \rho^d \in P^d_{loop} : \Omega_{G_{\mathcal{P}}}(B) \neq \emptyset \text{ with } \rho^d : \{B\} \implies \{H\}$$

Intuitively, there is at least one solution mapping for all rule bodies (i.e., BGPs) from the resulting graph of the evaluation of the linked program $\mathcal{P}$. Thus every rule in P^d_{loop} is guaranteed to be triggered.

Because the same tuples are generated repeatedly, the TripleQueue and RequestQueue are constantly re-filled with items. Thus, the program evaluation cannot be terminated e.g., by Algorithm 2. However, the existence of an execution loop cannot be understood as a design flaw of a given linked program, as it cannot be prevented in every case to achieve the goals of an application (see Example 15).

✎ **Example 15**

The linked program in Listing 8 has a starting graph that denotes the resource `acme:practiceTeam` is of type `p:Team` and establishes a `owl:sameAs` relation between `p:Team` and the concept of dbpedia for sports teams. The

[54] In Chapter 5 we employ the concept of subgraph pattern to match resource descriptions with search requests.

rules of the program together with the starting graph cause two execution loops:

1. Rule (1) causes every sports team to be retrieved. In particular the information resource of `acme:practiceTeam` returns the following graph:

   ```
   acme:player1 p:memberOf acme:practiceTeam .
   ```

   The returned triple causes rule (2) to infer that `acme:practiceTeam` is of type `p:Team`. In turn the retrieval of `acme:practiceTeam` is triggered by to rule (1) due to this inference. Thus rule (1) and (2) continue to be triggered alternately.

2. Rule (3) encodes the symmetry of `owl:sameAs` and is an example of a single rule that causes an execution loop: The rule infers that `db-owl:SportsTeam` is the same as `p:Team` and vice versa alternately.

Listing 8: Linked program that causes execution loops.

```
@prefix db-owl: <http://dbpedia.org/ontology/> .

# Starting graph
acme:practiceTeam rdf:type p:Team .
p:Team owl:sameAs db-owl:SportsTeam .

# (1) Lookup every team
{ ?u rdf:type p:Team . } => { [] http:mthd httpm:GET ;
                                 http:requestURI ?u . } .

# (2) Member of a team
{ ?m p:memberOf ?n . } => { ?n rdf:type p:Team . } .

# (3) Symmetry of owl:sameas
{ ?x owl:sameAs ?y . } => { ?y owl:sameAs ?x . } .
```

Removing duplicate tuples from processing can decrease the runtime of the system, as unnecessary derivations can be prevented. Duplicate tuples that are not just singular occurrences, but result from an execution loop have to be removed from processing to allow the program execution to terminate properly. The following mechanisms can be employed to remove duplicates from processing (compare [49, 119]):

- EquiJoin operators may directly filter out some duplicates, if the multimaps used in the Binding operators use distinct sets rather than lists to store arriving tuples. When the same tuple arrives multiple times at the same Binding operator of an EquiJoin operator the duplicate is removed automatically. The use of sets in the Binding operators does generally not to remove all duplicates from the processing, as not all triples are pushed to EquiJoin operators in the plan (see rule (3) in Listing 8). Thus, the use of distinct sets is not sufficient to guarantee the proper termination of a program with execution loops.

- We can use a distinct set to additionally store all triples that are pushed to the TripleQueue. If the set already contains an arriving triple, the triple is dropped, i.e. not put in the TripleQueue and thus removed from processing. The distinct set is sufficient to guarantee proper termination, as every triple has to pass the TripleQueue before processing. However, duplicates are not removed immediately once the duplicate is derived in the physical plan, thus the duplicates might still cause unnecessary derivations.
- We could use a distinct set at every operator in the plan to store every arriving tuple. If a set already contains an arriving tuple, the tuple is dropped. Thus duplicates are immediately filtered out after they are generated at any point in the physical plan and consequently a proper termination is guaranteed also in the face of execution loops.
- Finally, the approach to use a distinct set to store URIs of resources that have already been retrieved (including information about redirects) to prevent duplicate lookups, prevents duplicates from being processed (see Section 3.3.1). Strictly, duplicate tuples are not removed, but prevented as duplicate lookups are removed. To remove duplicate lookups guarantees termination of programs where all execution loops contain a request rule $\exists \rho^r \in P_{loop}$.

Using sets to keep track of duplicate requests and tuples requires more memory, but prevents unnecessary derivations. Additionally, managing the distinct sets, i.e., writing and reading of tuples, can also increase the runtime of the system, especially if there are not many unnecessary derivation that are prevented. Also, not every linked program necessarily causes an execution loop. Consequently, different combinations of the described mechanisms can be advantageous depending on the evaluated linked program.

> ✎ **Example 16**
>
> Consider a linked program that only retrieves a few initial resources $\mathcal{P} = (\emptyset, R, \emptyset, \emptyset)$ without further rules. The program only serves the purpose of evaluating a registered query over the retrieved data. There can be no execution loops and single occurrences of duplicates only appear if some of the retrieved resources contain the same triples. Therefore, the use of distinct sets to remove duplicates is unnecessary.

3.5 EXPERIMENTS

In this section we describe experiments as part of a systematic evaluation. We analyse the behaviour of a fully implemented system for the parallel evaluation of linked programs with the described push-based execution model. In particular, we use different threading models and analyse the system in terms of throughput with different degrees of parallelism, i.e., a different number of worker threads and cores. We also describe the processing time in experiments with different workloads, i.e., a different amount of triples and requests, and different sets of rules.

Our experiments are based on the Lehigh University Benchmark (LUBM) [44] and a synthetic tree dataset, to evaluate either mostly CPU-bound, mostly I/O-bound or mixed tasks. The LUBM benchmark evaluates the performance of 14 extensional queries over a dataset, which is scalable with respect to size, and an ontology from the university domain. While LUBM is a purely CPU-bound benchmark, we specifically use an additional synthetic dataset to be able to precisely control the size of the processed data and the delay of network requests, which allow us to evaluate the system behaviour under different conditions and identify the dependencies between network lookups and data processing.

We choose a tree as data structure of the synthetic dataset that can be traversed with recursion. The tree-shaped dataset is retrievable as distributed Linked Data resources, i.e., every node in the tree is a URI-identified information resource. We use numbers to denote individual nodes, with root node :1.

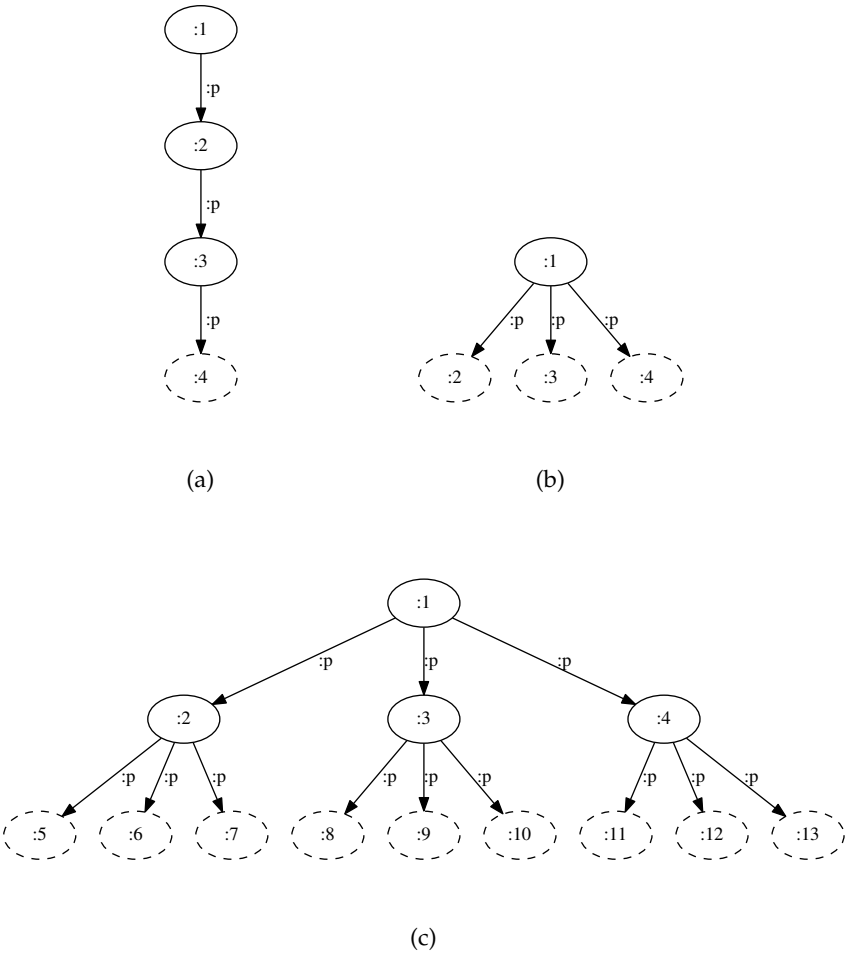


Figure 9: Synthetic tree dataset shapes: (a) a path ($d = 3, b = 1$), (b) a star ($d = 1, b = 3$), and (c) a tree ($d = 2, b = 3$). Leaf nodes are circles with dashed lines.

Parameter b specifies the breadth, and d specifies the depth of the completely balanced tree. Specifically, we denote b as the number of child nodes for every node, except for the leaves. We denote d as the length of the path from the root note to any leaf. Figure 9 shows some generated tree-shaped datasets, including

the two distinguished cases of a path-shaped dataset with $b = 1$ in Figure 9(a) and a star-shaped dataset with $d = 1$ in Figure 9(b).

When one of the nodes is retrieved, the returned triples represent the relation (i.e., a link) between the node and its children with a property `:p`. Consequently, the parameter b also specifies how many triples are returned when we retrieve a node, with the exception of leaf nodes: As leaves do not have any child nodes, a lookup on a leaf does not return any triples. However, a lookup on a leaf is possible with a response $(200, \emptyset)$.

✎ **Example 17**

We consider a tree with $d = 2$ and $b = 3$ as shown in Figure 9(c). A lookup $(\mathsf{GET}, \texttt{:1}, \emptyset)$ on the root node results in the response $(200, D)$, where D is the set of returned triples:

```
:1 :p :2 .
:1 :p :3 .
:1 :p :4 .
```

In Listing 9 we show the initial request (0) on the root node and the request rules (1) that specifies that the system will retrieve every child node by following the links. In some of the experiments we define the property `:p` to be symmetric with deduction rule (2) and transitive with deduction rule (3). With rules encoding symmetry and transitivity we materialise the transitive closure over `:p` from the synthetic dataset.

Listing 9: Rules used in experiments with synthetic tree dataset.

```
1
2  # (0) Initial request on the root node
3  { [] http:mthd httpm:GET ;
4     http:requestURI :1 . }
5
6  # (1) Request rule to follow all links to child nodes
7  { ?x :p ?y . } => { [] http:mthd httpm:GET ;
8                       http:requestURI ?y . }
9
10 # (2) Symmetry of :p
11 { ?x :p ?y . } => { ?y :p ?x . } .
12
13 # (3) Transitivity of :p
14 { ?x :p ?y .
15   ?y :p ?z . } => { ?x :p ?z . } .
```

We run experiments with the following hardware setups:

SETUP A A local machine with two Intel Xeon E5-2670 processors with 8 physical cores per processor at 2.60GHz and 250 GB main memory. Hyperthreading is enabled resulting in 32 logical cores. The operating system is Debian Wheezy 64bit GNU/Linux with 3.2.0 kernel.

SETUP B An Amazon Web Services c4.8xlarge instance with two Intel Xeon E5-2666 v3 processors with 9 physical cores per processor at 2.90GHz and

60GB main memory. Hyperthreading is enabled resulting in 36 logical cores. The operating system is Ubuntu 14.04.2 LTS 64bit GNU/Linux with 3.13.0 kernel.

SETUP S Additionally, in experiments that involve network lookups we use a server on which the resources of the synthetic tree dataset are deployed: A virtual machine with 4 logical cores based on a AMD Opteron 6344 processor at 2.60GHz and 16GB main memory connected via 1Gbit/s Ethernet. The operating system is Debian Wheezy 64bit GNU/Linux with 3.2.65 kernel.

Our system for the evaluation of linked programs is implemented in Java, and the experiments are executed on OpenJDK 1.7u24[55]. We use Java's ConcurrentLinkedQueues for the queues, ConcurrentHashMaps as multimap implementation, with ConcurrentLinkedQueues as tuple lists in value position of the multimaps[56], and a ConcurrentHashMap as basis for a distinct set to remove duplicates at the TripleQueue (see Section 3.4.4). Thus, we are guaranteed to break execution loops, and the programs terminate.

We experiment with different threading models: *serial*[57], *rounds*, *spinning* and *blocking*. We implemented the blocking threading model with a single TripleQueue and RequestQueue (*blocking single*), as well as separate queues for each worker thread (*blocking multi*).

3.5.1 *CPU-bound Tasks*

We use the LUBM benchmark as a completely CPU-bound task to evaluate the performance of our system while answering the queries of LUBM, where deduction rules materialise the necessary inferences. In particular, we use the OWL LD rule set[58] directly, as well as a rewritten rule set, custom tailored for LUBM to generate the inferences. The employed custom rule set for LUBM can be found in Appendix A.3 in Listing 19. We first generate all input triples, before applying a rule set.

Figures 10(a) and 10(b) show the throughput on LUBM(200) on Setup A and B respectively with the custom rule set, depending on the number of employed threads. A detailed list of results can be found in Appendix A.2 in Tables 15 and 16.

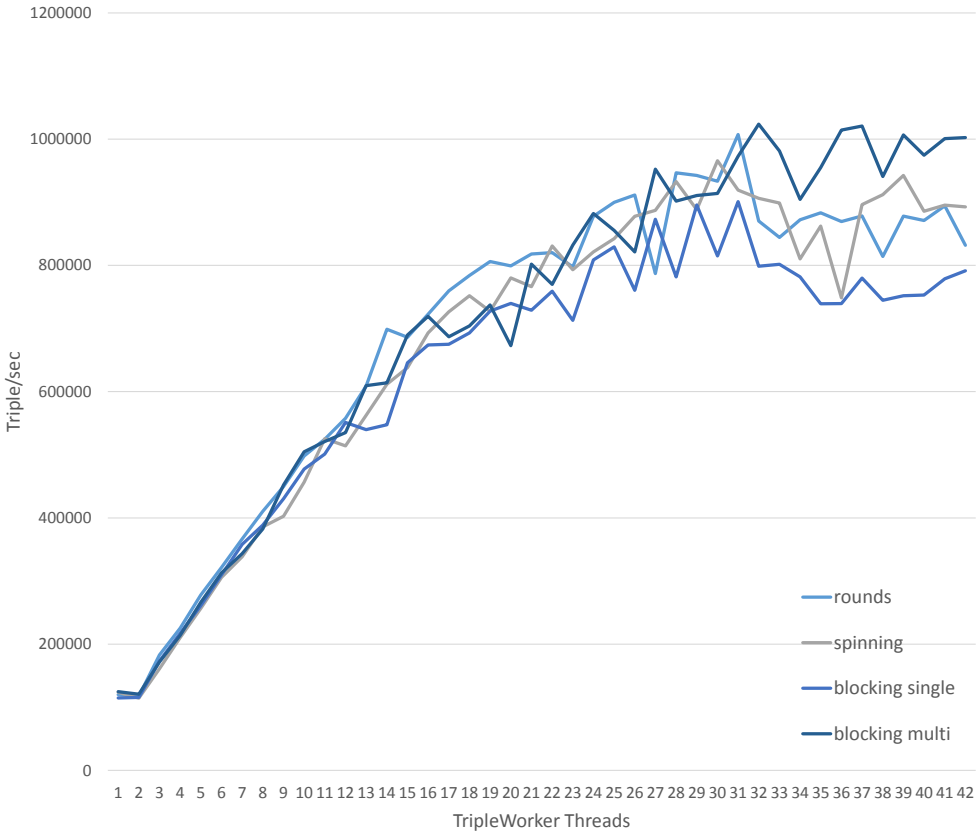
In general we see a decreasing speedup with an increasing amount of employed threads. E.g., on average (over all threading models) adding one thread between 1-8 threads on setup A increases the throughput by 38 715 triple/s. Between 9 and 32 threads adding one thread increases the throughput by 21 169

[55] We use the following Garbage Collector settings: `-XX:+UseParallelGC -XX:+UseParallelOldGC`.

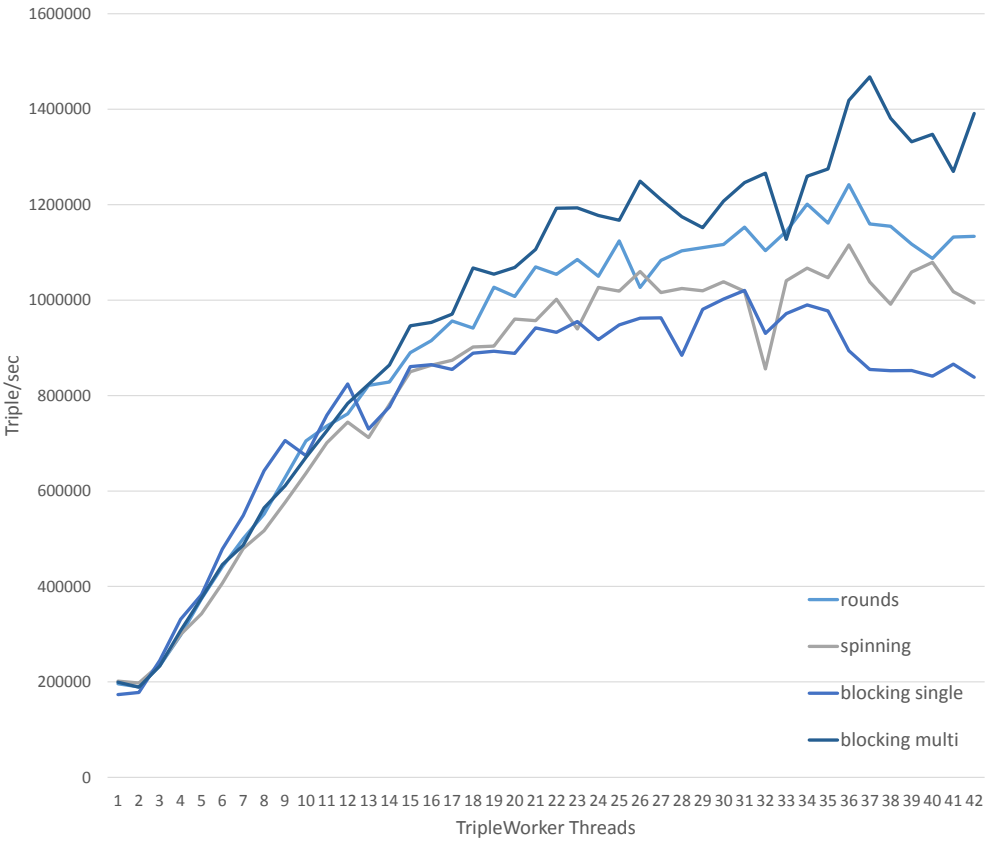
[56] In early experiments, ConcurrentLinkedQueues turned out to use less memory and to be faster then ConcurrentHashMap (as basis for sets), despite not removing duplicates (cf. Section 3.4.4).

[57] The *serial* model is only partially comparable, as it does not allow to increase the number of threads.

[58] `http://semanticweb.org/OWLLD/`, retrieved 2015-04-10.

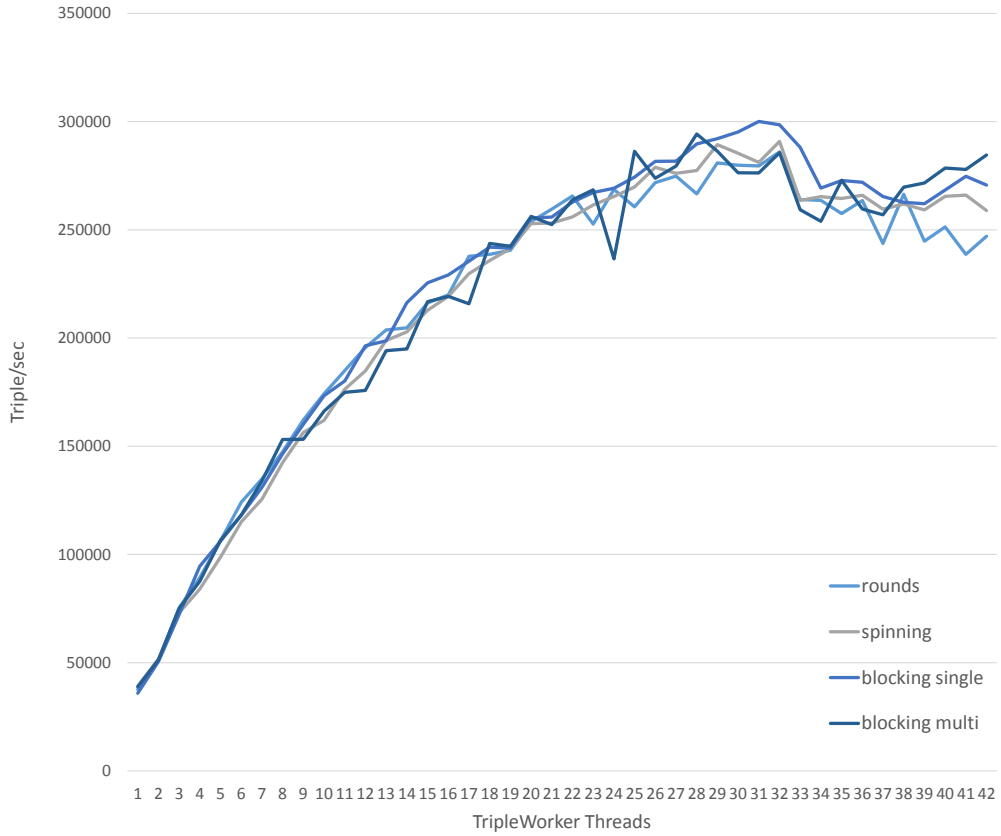


(a) Setup A

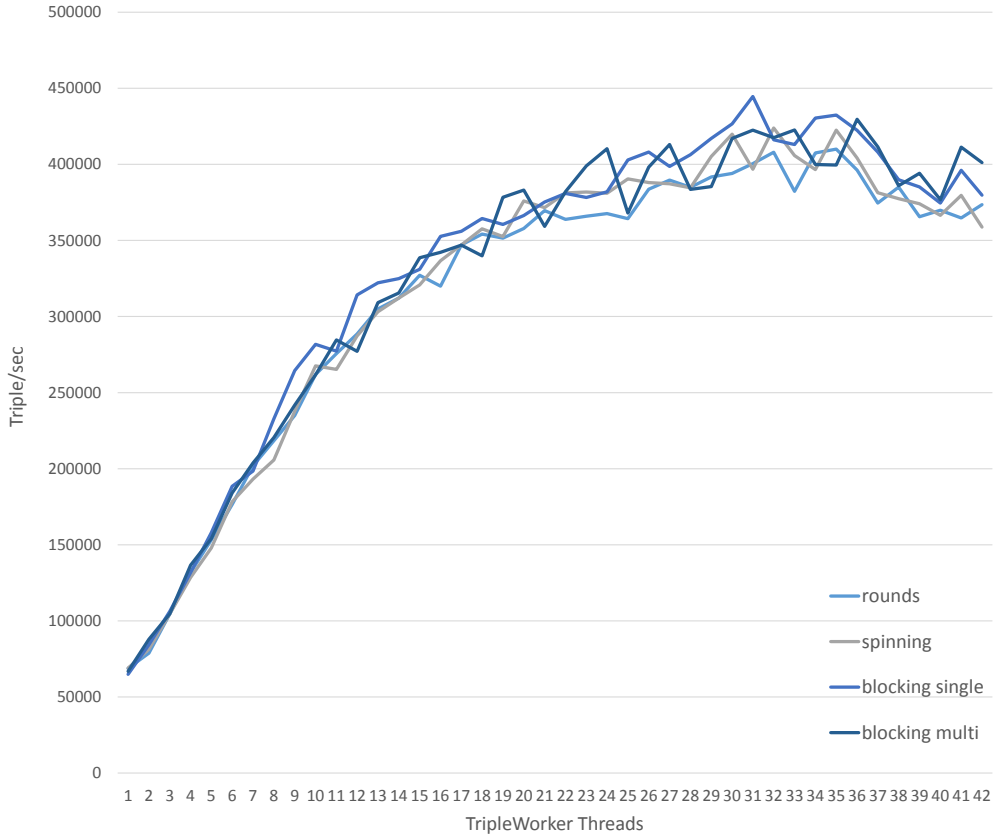


(b) Setup B

Figure 10: Throughput of of parallel processing of LUBM 200 with custom rule set.



(a) Setup A



(b) Setup B

Figure 11: Throughput of of parallel processing of LUBM 200 with OWL LD rule set.

triple/s and between 33 and 42 threads adding an additional thread even decreases the throughput by -2 023 triple/s. Similarly on setup B on average adding one thread between 1-9 threads increases the throughput by 62 518 triple/s. Between 10 and 36 threads adding one thread increases the throughput by 22 378 triple/s and between 37 and 42 threads adding an additional thread decreases the throughput by -3 254 triple/s.

One reason for these diminishing returns is described in Amdahl's Law [3], as the reasoning tasks in LUBM cannot be completely parallelised [80] and only the part of the system that can be parallelised is optimised by increasing the amount of employed threads. Additionally, an increase of threads beyond 32 threads on setup A and 36 on setup B decreases throughput, as only 32 and 36 cores are available on the respective machines. The completely CPU-bound task leverages the system resources fully with 32 threads and adding more threads does not increase the degree of parallelism, but rather adds unnecessary overhead.

When comparing the individual threading models, we see on both setups that the differences between the models increase with an increase in employed threads. We see that the *blocking single* approach results in a generally lower throughput compared to the other approaches. The overhead of the *rounds* and *spinning* approach are of minor importance for the LUBM benchmark, as the large set of input triples is generated before the processing starts. Therefore, the threads are rarely in a sleep state, which results in a greater effect of the overhead of directly managing the thread state in the *blocking single* approach. However, *blocking multi* generally performs best, achieving about 1 mio triple/s on setup A and 1.4 mio triple/s on setup B, as the advantage of removing the bottleneck of having a single TripleQueue and RequestQueue mitigates the overhead. The *serial* approach is not included in Figure 10, as it cannot be used with more than one thread. On setup A *serial* reaches 244 008 triples/s and on setup B 434 267 triples/s, thus *serial* performs better than other threading models with one thread, because there is no unnecessary coordination overhead.

Figures 11(a) and 11(b) show the throughput on LUBM(200) on Setup A and B respectively with the OWL LD rule set, depending on the number of employed threads. A detailed list of results can be found in Appendix A.2 in Tables 17 and 18.

We see a similar behaviour of diminishing returns when employing the OWL LD rule set compared to the custom rule set. However, the individual threading models exhibit almost no differences. The more complex OWL LD rule set causes an overall longer processing time with a larger amount of solution sequences stored in the multimaps of the Binding operators in the physical plan. As individual processes executed in a thread require more time to be executed (e.g., due to longer read and write times in the larger multimaps), the differences of the threading models are largely mitigated.

Figures 12 and 13 show again the throughput of LUBM(200) on setup A and B with a custom rule set and the OWL LD rule set, depending on the number of employed threads. Here however, we only employ the same amount of cores as TripleWorker threads and deactivated the other cores for the individual runs. If we use more threads than the maximum amount of cores available on the system,

we use all cores. [59] E.g. if we use 8 threads on setup A (32 cores) we deactivate 24 of the 32 cores of the machine; if we use 33 threads, we use all available 32 cores. A detailed list of results can be found in Appendix A.2 in Tables 19–26.

Please observe the following vertical markers:

1. Up to 8 threads on system A and 9 threads on system B, all employed threads run on the cores of the same processor, as the cores of the other processor are deactivated. Above 8 threads on system A and above 9 threads on system B, the threads run on the cores of both processors. Data of the running processes has to be transferred over the Quick Path Interconnect (QPI)[60] of the mainboard, as the operating system switches threads between the cores on different processors.

2. Above 16 threads on system A and 18 threads on system B, the processes are not guaranteed to be executed on a dedicated physical core, but share partially share physical cores via hyperthreading.

3. Above 32 threads on system A and 36 threads on system B, we fully leverage all available logical cores of the systems and just increase the amount of employed threads.

We show the average throughput out of 5 runs and provide error bars indicating the standard error from those five runs.

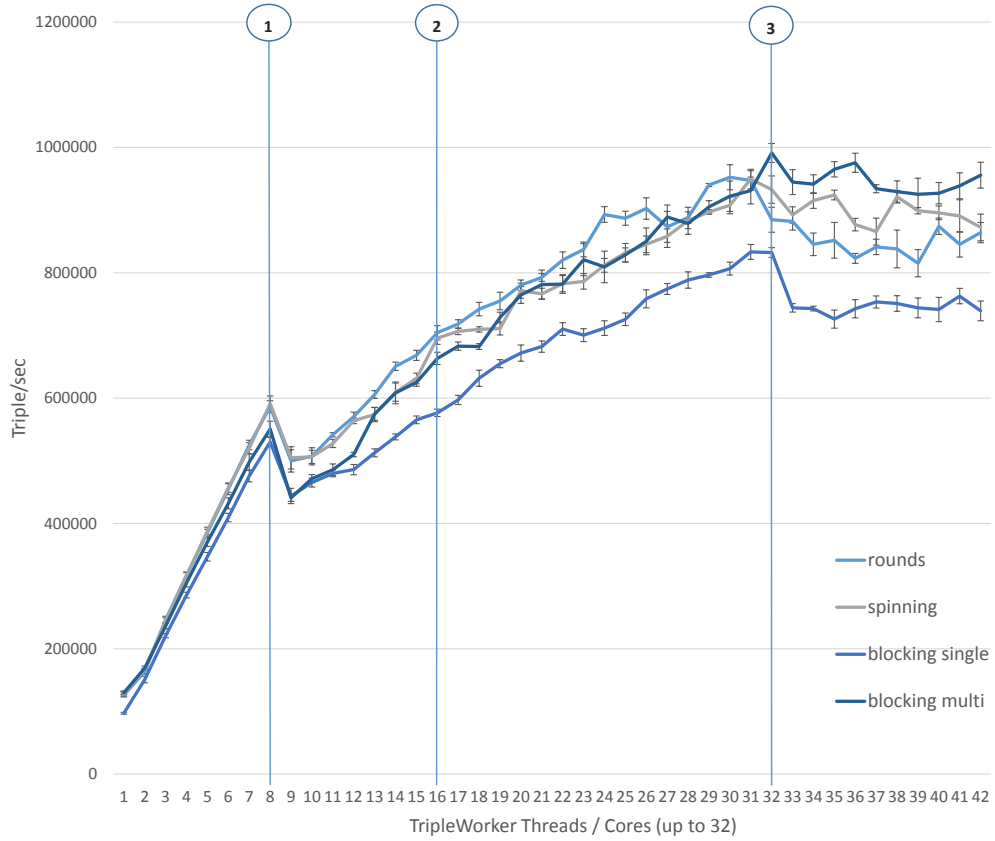
We see again a decreasing speedup with an increasing amount of threads and cores. However, the speedup before the marker 1 is considerably higher compared to the experiments without deactivated cores. Between marker 1 and 3 the speedup is lower compared to the experiments without deactivated cores. E.g., with custom rules on average (over all threading models) adding one thread between 1-8 threads on setup A increases the throughput by 63 475 triple/s. Between 9 and 32 threads adding one thread increases the throughput by 13 780 triple/s and between 33 and 42 threads adding an additional thread decreases the throughput by -5 239 triple/s. Similarly with custom rules on setup B on average adding one thread between 1-9 threads increases the throughput by 81 618 triple/s. Between 10 and 36 threads adding one thread increases the throughput by 9 756 triple/s and between 37 and 42 threads adding an additional thread decreases the throughput by -10 175 triple/s.

The reason for the different behaviour is that the overhead of using QPI on both systems is avoided before marker 1, and comes only into effect when employing more then 8 cores on setup A and 9 cores on setup B. We can especially observe that the throughput decreases shortly after marker 1, as 1-3 additional cores does not make up for the overhead introduced by QPI. The decrease of throughput occurring on two independent setups when employing slightly more cores than available on one processor supports the conclusion that the effect is related to the QPI.

We also see a tendency that the standard error increases with the amount of cores implying a higher variance of the throughput when more cores are

[59] We use Linux `taskset` to specify the number of cores on which processors used for the JVM.

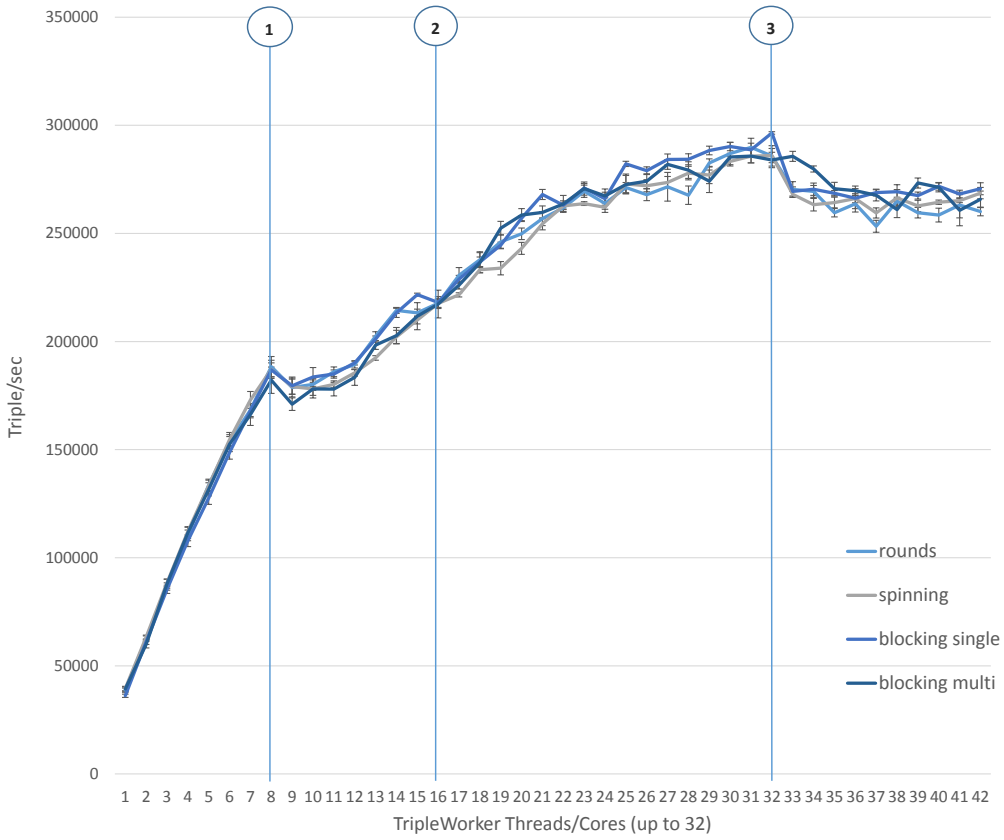
[60] http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html



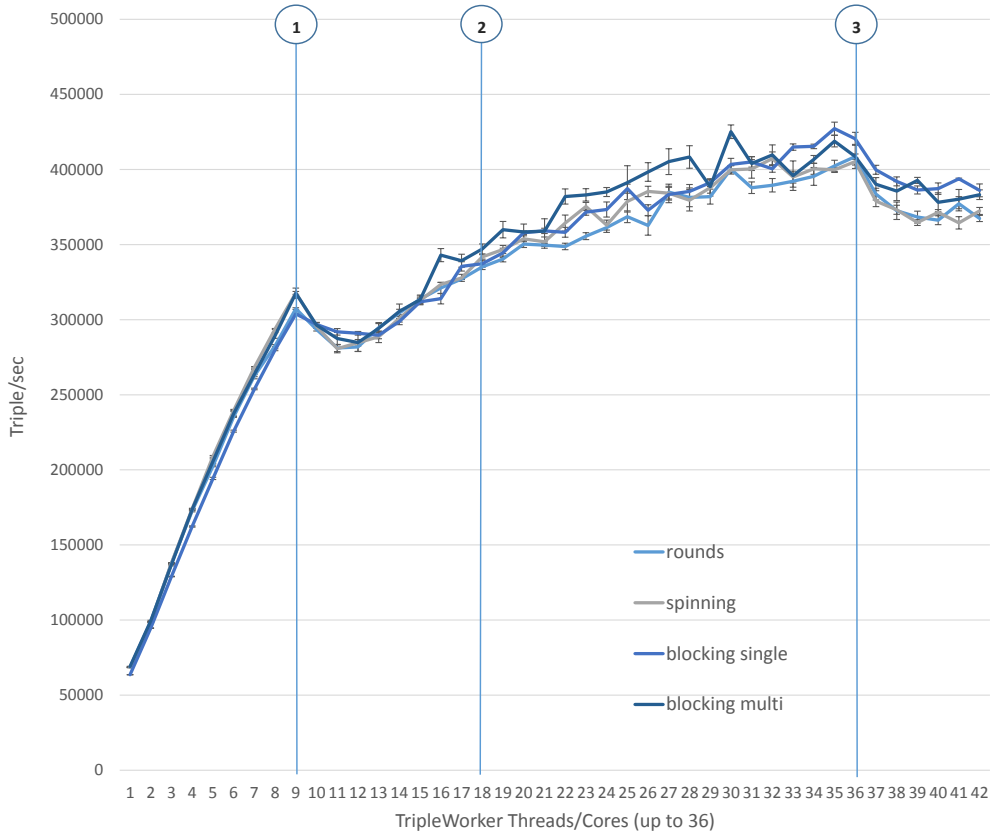
(a) Setup A

(b) Setup B

Figure 12: Throughput of of parallel processing of LUBM 200 with custom rule set; number of cores adapted to number of threads (average of 5 runs).



(a) Setup A



(b) Setup B

Figure 13: Throughput of of parallel processing of LUBM 200 with OWL LD rule set; number of cores adapted to number of threads (average of 5 runs).

employed. Table 5 and 6 show the average standard error we measured for Lubm(200) with custom rules between the markers for the two systems respectively. However, increasing the amount of employed threads with the same number of cores (e.g., above 32 threads with system A) does not further increase the standard error. The operating system assigns threads to cores randomly. As more cores are available, there are more possibilities for assignments of threads to cores. E.g., a thread might constantly be assigned to cores on the same processor, thus avoiding data transfer between processors via QPI; a thread might be constantly switched between cores on different processors, thus increasing the amount of data transferred via the QPI.

The differences of the threading models are similar to the experiments without deactivated cores.

Table 5: Average standard error from 5 LUBM runs for the throughput between the marked amount of threads for system A.

between threads:	1-8	9-16	16-32	32-42
rounds	6201	9295	11638	18203
spinning	6159	10500	11863	13787
blocking single	5578	7026	8781	12503
blocking multi	7311	8529	15088	17018

Table 6: Average standard error from 5 LUBM runs for the throughput between the marked amount of threads for system B.

between threads:	1-9	10-18	19-36	37-42
rounds	2343	10944	17601	16235
spinning	2517	8191	14178	12751
blocking single	2745	8264	9027	10078
blocking multi	2069	10409	29883	28138

Next, we experiment with the synthetic tree dataset in a completely CPU-bound setting. We use a tree datasets with breadth $b = 2$ and increasing depth d, and thus an increasing amount of triples. The triples are all locally generated, i.e., no resources have to be retrieved via HTTP, but the complete graph is locally available as initial graph. We employ a program consisting of deduction rules for symmetry and transitivity (2) and (3) in Listing 9 and measure the required processing time until the fixpoint is reached on setup A. As the data is generated locally, rather than retrieved from the server, the program evaluation is completely CPU-bound. Figure 14 shows the results, with the amount of processed triples including derived triples (i.e., the amount of triples in the result

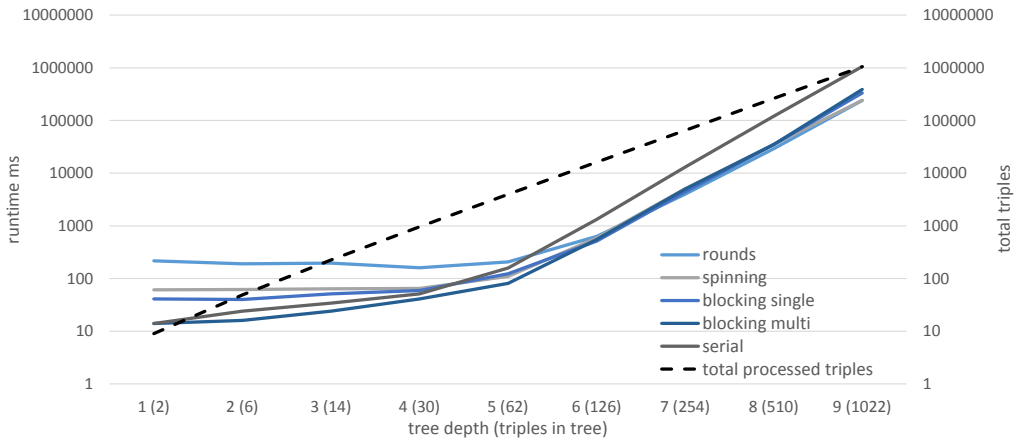


Figure 14: Runtime to process locally generated synthetic tree dataset with rules for symmetry and transitivity depending on tree with breadth $b = 2$ and increasing depth.

graph of the program) on a secondary horizontal axis for comparison. We fix the amount of TripleWorker threads to 32. A detailed list of results can be found in Appendix A.2 in Table 29.

We see that the different threading models with the exception of *serial* exhibit differences only up to a tree depth of 5 (3 969 processed triples). Below depth 5 the triples arriving in the TripleQueue can almost instantly be taken and processed by a TripleWorker. Beyond depth 5 the processed data in the operator tree becomes large enough that the read and write operations in the larger multimaps become a dominant factor for the processing time. Consequently the differences in the threading models become less important with increasing number of triples.

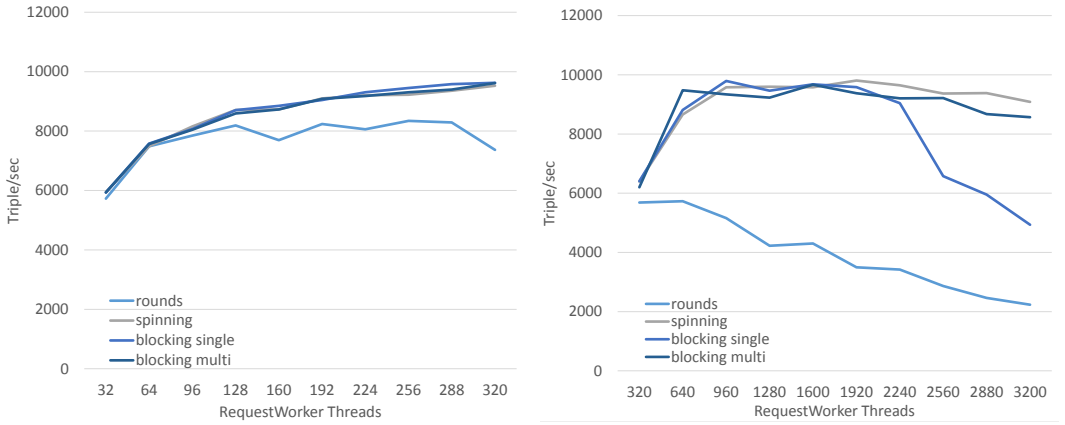
The baseline *serial* model is faster than the other multi-threaded models below trees of depth 4 (961 processed triples), with the exception of the *blocking multi* model, indicating that the data size is to small to sufficiently leverage parallel processing to make up for the overhead of most of the multi-threaded models. The *blocking multi* model, however, shows consistently the shortest runtimes.

The experiment shows that in a CPU-bound setting below a certain dataset size to process only *blocking multi* would be preferable over *serial*. Above a certain dataset size any multi-threaded model is preferable over *serial*.

3.5.2 *I/O-bound Tasks*

Next we run experiments that contain predominantly I/O-bound tasks. We retrieve the information resources of a synthetic tree dataset with breadth $b = 6$ and depth $d = 6$, which results in a total graph with 55 986 triples and resources[61], from the server (setup S). We do not use any derivation rules to calculate symmetry or transitivity of the property. Thus, we execute a linked program

[61] The amount of triples and information resources in the dataset is equivalent, as for every resource there is exactly one triple linking to a it.



(a) Between 32 and 320 RequestWorker threads. (b) Between 320 and 3200 RequestWorker threads.

Figure 15: Throughput of system retrieving a tree dataset without deduction rules with breadth $b = 6$ and depth $d = 6$ depending on number of request threads; data is retrieved by network requests with about 2ms delay per request.

on setup A with the initial request (0) and rule (1) in Listing 9, which is I/O bound, as the tasks to perform mostly consist of network lookups.

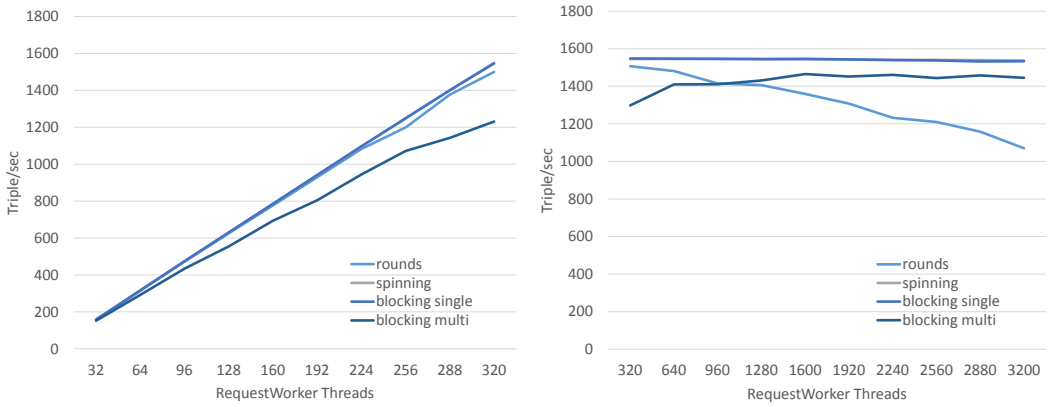
The program retrieves the root note as initial request and then follows all the links until the complete graph is retrieved. Leaf nodes are also retrieved, albeit leaves do not return any triples. On average a HTTP request on a node takes 2 ms, which also holds true for the leaf nodes, even though a request on a leaf does not include any data transfer.

Figure 15 shows the achieved throughput of the system for the processing of the linked program depending on the number of employed RequestWorker threads. In particular, Figure 15(a) shows values between 32 and 320 employed threads and Figure 15(b) shows values between 320 and 3 200 employed threads. As every retrieved triple causes a distinct request, the shown values for triple/s are equivalent to the achieved request/s value. A detailed list of results can be found in Appendix A.2 in Table 27.

As the program evaluation is I/O-bound, we see an increase in throughput when increasing the number of request thread well beyond the number of available cores. While a thread that executes a request waits for the response, the operating system can switch another thread onto the core of the waiting thread. Thus, more requests can be executed in parallel than there are cores available on the setup[62].

The effect of using a large number of threads is less prevalent in the *rounds* model, as towards the end of every round only a fraction of the threads stay active and the others just cause overhead. Consequently, the *rounds* model exhibits only a speedup with up to 288 request threads. Adding more threads decreases throughput due to overhead for managing more threads.

[62]For the experiments we deactivated the politeness policy that not more than one request per host can be executed in parallel. In a real setting the parallel requests can be executed on resources from multiple hosts.



(a) Between 32 and 320 RequestWorker threads. (b) Between 320 and 3200 RequestWorker threads.

Figure 16: Throughput of system retrieving a tree dataset with rules for symmetry and transitivity with breadth b = 6 and depth d = 6 depending on number of request threads; data is retrieved by network requests with about 200ms delay per request.

The other models show a speedup with up to 1920 request threads. Beyond that the throughput decreases, as the overhead of managing additional threads outweighs the benefits of parallelism. The trade-off effect is especially visible for the *blocking single* model, where the overhead of directly controlling the RequestWorker states is not mitigated by resolving the bottleneck of having only single queues.

The highest achieved throughput is about 10000 triple/s (or requests/s) for all multi-threaded models except *rounds*. For comparison, the baseline model *serial*, which can only employ one thread reaches in the same experiment only 873 triple/s. However, using the approach to dynamically adapt the amount of RequestWorker threads, as described in Section 3.4.3 with the *blocking multi* model, the system achieves a throughput of 11 151 triple/s. We use a threshold of 400 ms for the maximum expected worktime of a RequestWorker. The adaptive adjustment of threads is especially in a scenario like the tree dataset advantageous, where the amount of available requests to perform in the RequestQueue varies strongly over the processing time: After the first request to the root node only 6 links are found (i.e., 6 requests), however every retrieved resource adds 6 new requests, thus the rate at which resources to retrieve are identified increases over time. The adaptive adjustment of threads is able to scale the amount of employed threads over the course of the processing, thus avoiding overhead of too many threads in the beginning and increasing the amount of threads towards the end as necessary.

Next we run an experiment on the same synthetic tree dataset (b = 6, d = 6), but simulate a delay of 200 ms for every request. The delay causes the effect of I/O-bound tasks to be more prevalent as shown in Figure 16. A detailed list of results can be found in Appendix A.2 in Table 28. We see a larger speedup when increasing the number of request threads compared to the experiment without simulated delay, as the longer delay time allows to perform more requests simultaneously. As the requests take generally longer, the *rounds* model also profits

longer from increasing the number of threads, because the individual rounds take more time and the down time towards the end of the round does not dominate the overall runtime.

We also see that the approach to use multiple queues in the *blocking multi* model has an adverse effect: With the longer delay the chances that one of the RequestQueues is briefly empty increases, which causes the algorithm to check whether the fixpoint is reached more often (see Algorithm 2), thus introducing overhead. In contrast, the other models use only one RequestQueue, thus the single queue is rarely empty, leading to fewer termination checks. With the delay the top throughput, the system achieves id about 1 500 triple/s.

For comparison, the approach with a dynamic adjustment of RequestWorker thread achieves 3024 triple/s. The positive effect of scaling the amount of threads is increased when delay is introduced for the requests, as the individual active RequestWorker, reach their threshold faster with only a few requests in their queue. Thus the system can react more precisely on the amount of requests and by increasing of decreasing the amount of employed threads more quickly.

In summary we see that a large amount of threads beyond the number of available cores for the I/O-bound tasks is advantageous, especially if the delay causes the input rate to be slow. However, the overhead of introducing more threads can outweigh the benefits of the increased parallelism, with different threading models causing different amounts of overhead. Dynamically adjusting the amount of employed threads to execute requests is a valid approach to mitigate the overhead and allows the blocking multi model to perform fastest.

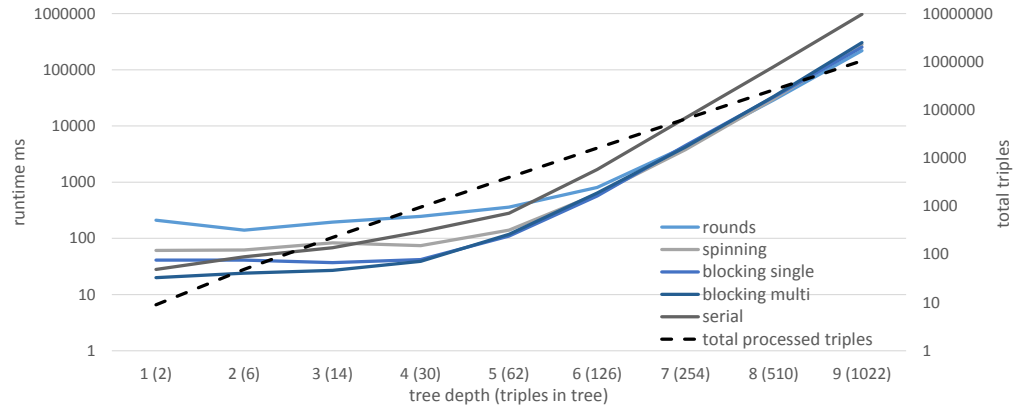
3.5.3 *Mixed CPU- and I/O-bound Tasks*

Finally, we combine the retrieval of the synthetic tree dataset from the server (setup S), with the application of rules for the symmetry and transitivity on setup A. Consequently, we execute the linked program with the initial request and all rules in Listing 9. Thus, in the following experiments we require TripleWorker (fixed to 32) and RequestWorker (fixed to 64) threads, combining CPU-bound and I/O-bound tasks.

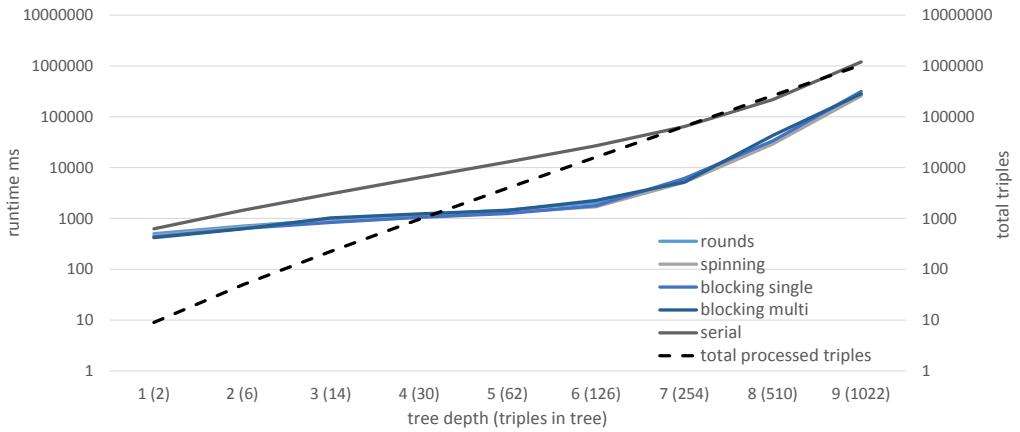
We again use a tree with breadth $b = 2$, increase the depth of the tree, and measure the absolute runtime to retrieve all resources via link traversal and materialise all triples resulting from symmetry and transitivity of property :p. Figure 17(a) shows the results without artificial delay and Figure 17(b) shows the results with a simulated delay of 200 ms. A detailed list of results can be found in Appendix A.2 in Table 31.

Without artificial delay the *blocking multi* model performs consequently best, but larger differences between the threading models are only visible below a tree depth of $d = 5$, with the exception of *serial*. As the resources of the tree can be retrieved with almost no delay the behaviour of the system in terms of runtime is similar as in the experiment with a locally generated tree dataset (see Figure 14).

However, the I/O-bound tasks in the experiment with no delay influence the runtime to some extent: Compared to the completely CPU-bound experiment, the baseline *serial* model does not outperform the *spinning* and *blocking single* model equally well under small tree sizes. Only with a tree of depth $d = 1$ (i.e.,



(a) 2 ms delay per request



(b) 200 ms delay per request

Figure 17: Runtime to process retrieved synthetic tree dataset without derivation rules depending on tree with breadth $b = 2$ and increasing depth; data is retrieved via network requests.

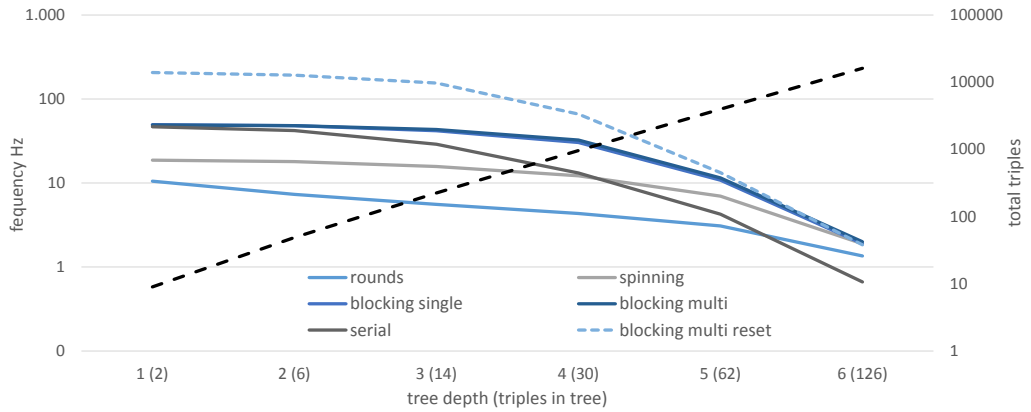


Figure 18: Average frequency of 1 000 repeated runs of a linked program retrieving a tree dataset with breadth $b = 2$ and increasing depth and materialising triples for symmetry and transitivity.

2 triples), *serial* remains the second best approach, as parallel processing is not required. However the *rounds* approach still suffers from the downtime at the end of the on average shorter rounds, with smaller tree sizes.

In the experiment with 200 ms delay per request, the multi-threaded models exhibit almost no difference, indicating that the I/O-bound tasks dominate the overall runtime. In turn the differences in the experiment with no delay are caused by the different performances of the TripleWorkers. The *serial* model requires at least the runtime of the summed up delay of all requests, which is why it performs considerably slower than the other models with a delay.

Lastly, we run an experiment for a scenario where a linked program has to be repeatedly executed in fast succession, e.g., to constantly probe if Web resources have changed (see Example 12). We again execute the linked program shown in Listing 9, including retrieval of resources and deduction rules for symmetry and transitivity. However, once the system reaches the fixpoint, i.e., is finished processing and the complete result graph is materialised, we start the processing of the program again. We repeat the execution of the program 1 000 times and measure the average frequency with which we can run the program. Results are shown in Figure 18. A detailed list of results can be found in Appendix A.2 in Table 30.

In correspondence to the absolute runtimes in Figure 17(a), we see that *rounds* achieves the lowest frequencies and *blocking multi* the highest frequencies. E.g., for a tree dataset with $b = 2$ and $d = 4$, with results in 30 retrieved triples, the system calculates the result graph containing 961 triples with 31 Hz using *blocking multi*. The *serial* approach performs with lower frequencies compared to the multi-threaded models with an increasing tree size.

When the fixpoint of the processing is reached and the program terminates, all worker threads are shut down. Therefore, if the processing has to start again, new threads have to be initiated in the system. We extended the *blocking multi* threading model to not shutdown the threads after the fixpoint is reached, which is specifically possible as we control the state of the threads directly in the *blocking multi* model. Rather than shutting the threads down, the physical operator

plan is cleared from intermediate results, i.e., the multimaps in all Binding operators are discarded and all distinct sets are emptied. To restart the program the initial request on the root node is performed again[63], which triggers again the same processing. With this approach, denoted as *blocking multi reset* in Figure 18 the system is capable to outperform the other threading models for all tree sizes smaller than $d = 6$. E.g., for a tree dataset with $b = 2$ and $d = 4$, which results in 30 retrieved triples, the system calculates the result graph containing 961 triples with 66 Hz using *blocking multi reset*.

Above a tree depth of $d = 6$ the achieved frequencies of the individual threading models converge (with the exception of serial), as the overall runtime of a singe program run is large enough to mitigate individual differences of the models with respect to execution frequencies.

Overall the experiments provide evidence that our proposed execution model has advantages in the presence of I/O-bound tasks, even when the performed tasks are predominantly CPU-bound. In particular the capability to scale the number of employed threads for I/O-bound and CPU-bound tasks independently emerges as important, as both thread pools can be separately adjusted to achieve a higher throughput before the introduced overhead outweighs the benefits.

Intertwining I/O-bound and CPU-bound tasks allows to conflate linked programs to a mostly CPU-bound problem, if enough data processing tasks have to be performed. Thus, the network requests are done in parallel to the processing without interfering with the data processing.

The choice of the used threading model can have a sizable impact. Specifically, the overhead of managing threads directly is only worthwhile, if few I/O-bound tasks are present in relation to the CPU-bound tasks. However, the introduced overhead can be mitigated, if the direct control is leveraged for the use of multiple queues, which removes a bottleneck in the processing.

3.6 RELATED WORK

There are many approaches for link-traversal query processing. In their work Umbrich et al. [118] study the impact on the recall of query results when taking into account reasoning constructs. We describe methods to carry out rule-based reasoning in combination with HTTP requests. In contrast to [30], who specify path expressions on queries, we only allow for processing of BGP queries. However, we can specify link expansion in a more fine-grained manner using request rules.

Our work also differs from approaches to answer queries over multiple data sources using views [66], as we do not assume to have source descriptions that provide a mapping of the local source schemata to a global schema. We allow to follow links during evaluation without prior knowledge about the retrieved content until a lookup is performed.

[63] The system would also include an initial graph to restart processing, if one is defined in the linked program.

In [108] we propose a system that also takes into account data manipulation (CRUD) operations, which will be the basis of our extension to programs that allow to change resources in Chapter 4. Here, we focus on read operations and on the parallel execution, specify optimisations, and conduct extensive performance evaluation.

Our algorithm to calculate joins in the evaluation plan described in section 3.3.3 is based on the symmetric hash join operator as proposed by Wilschut and Apers [128]. To enable parallel processing Wilschut describes a pipeline execution model, where every operator is executed by a process. In the pipeline model data is "pushed" from one operator to another by inter-process communication (i.e., pipes).

Ladwig and Tran [62] also propose a push-based based model where every operator is executed in its own process. Ladwig argues that the resulting streaming architecture, where data is streamed between the running processes is specifically advantageous, if at least some data sources are unknown before the query evaluation starts. However, they implement their system using the Scala actor model, with a queue and a lightweight thread for each operator, and do not consider rules.

Graefe [41] argues that the overhead introduced by the inter-process communication can be avoided by employing an execution model that allows operators to schedule each other within a single process. In Graefes model operators can request new data items from other operators whenever new data items are required. The operators iterate over the received (pull) data items to produce new (intermediate) results. Thus, Graefe also eliminates the need to buffer data in pipelines. However, Graefe points out that in a scenario where data-sources have to unload data as it arrives, a more data-driven push-based model might be more appropriate.

Hartig et al. [48] also use a demand-driven (iterator) model for query evaluation, with provisions for delaying/postponing requests to other operators.

In our execution model operators also schedule each other within a single process, but in a data-driven push-based manner: Every operator iterates over its intermediate results and pushes the data items to subsequent operators within the same process. Thus, we avoid the overhead of inter-process communication as well as cater to a scenario where new data from network lookups can continuously processed.

Motik et al. [76] also propose a parallel system for general recursive Datalog rules, where multiple processes extract facts from a database and evaluate the facts over sub-queries obtained from the rules. The result of the evaluation is written back to the database and consequently available for the evaluation of other sub-queries. The approach of Motik et al. relies on the availability of a storage scheme that allows for efficient evaluation of the sub-queries (i.e., indexes over the facts in the database) and efficient update mechanisms. In our approach, the data is fetched from the network, and triples are directly pushed through the operators which maintain multimaps for joins.

Also motivated by the goal to avoid overhead from process scheduling and inter-process communication Carney et al. [22] propose an execution model with an active scheduler. Here sequences of operators are stored in a queue. A router

assigns data items to the queue elements and a scheduler that monitors available system resources decides which queue elements (i.e., operator sequences) are executed in a process. The active scheduler is comparable to the coordination approach in our approach described in Section 3.4. However, we do not queue specific operator sequences to be executed, as the scheduling of operators is data-driven, i.e., data items are pushed from one operator to another. Further our approach does not actively monitor available system resources, but rather employs as many evaluation worker threads as there are cores available to optimally leverage system resources. In the scenario with network requests that we focus on, we optimise the amount of request worker threads with respect to the latency of remote data sources, as the network latency represents the bottleneck of the system.

Many approaches rely on an a priori fixed T-Box. Thus, a strand of work in rule-based reasoning revolves around the idea of separating T-Box (triples with `rdf:type` property and those with RDF, RDFS and OWL vocabularies) and A-Box. Hogan et al. [53] were the first to use the fact that many rule sets consist of joins between T-Box statements and A-Box statements. Their system holds T-Box in memory, and does scans over A-Box triples to perform the joins. Similarly, [119] uses specialised algorithms tailored to a particular rule set. In contrast, we provide a general rule processor for positive Datalog on triples, that allows e.g., for arbitrary join operations over instance data. Heino and Pan [49] propose a parallel reasoning system based on a shared memory architecture that avoids memory and communication overhead of duplicate results during evaluation. The reasoning system of Heino uses individual compute kernels for different rules. An instance of a compute kernel (i.e., a thread) evaluates the result of a rule for a single instance triple, based on separation of A-Box and T-Box. The approach of Heino requires no rule to depend on more than one instance triple, as the communication of entailment results is only performed between the evaluation of a complete rule.

Grosof et al. [42] note that rules can be rewritten in the face of T-Box statements, yielding more efficient rule sets. The drawback for using the T-Box to optimise rule processing is that the T-Box needs to be known (and fixed) before evaluating the query plan. Given that we aim for a general-purpose link-following, in which the processor might access hitherto unknown T-Box statement during query evaluation, the optimisation does not apply in our scenario.

3.7 SUMMARY AND FUTURE WORK

In this chapter we have presented methods to access, integrate and query Linked Data in an integrated system for the use in the development of Web-based applications. In particular, we have shown how the application logic can be specified with linked programs consisting of a set of declarative rules.

The specifications include request rules, to take into account the links between resources to be able to discover new resources in a decentralised environment, and deduction rules, to specify the semantics of data items as defined in knowledge representation languages such as RDFS and fragments of OWL. Request rules that allow to define the expansion of links can be defined in a fine-grained

manner, thus avoiding unnecessary requests. Deduction rules allow to define the semantics that has to be taken into account for the alignment of retrieved resources, leveraging interlinked Web-accessible schemata on-the-fly. Additionally, queries can be specified to extract the required information from the materialised graph. By providing syntax and semantics of the rules we have answered Research Question 1.1.

We have introduced a system for the parallel evaluation of the cyclic plans resulting from the rules and queries, built upon a push-based execution model. The push based execution model allows to separate threads for the evaluation of the plan (i.e., CPU-bound tasks) and the execution of network requests (i.e., I/O-bound tasks). Thus, the system can be configured to leverage system resources adjusted for the combination of both CPU-bound and I/O-bound tasks. We also introduced an approach for the adjustment of the amount of employed threads at runtime to react to the dynamically discovered amount of required network lookups. Consequently, the proposed execution model answers Research Question 1.2: The model caters specifically for data processing scenarios with network lookups, as it is capable of including interlinked schema information discovered at runtime for the data integration.

We have answered Research Question 1.3 by describing and evaluating several threading models that implement the push-based execution model for the parallel evaluation of rules and queries. In particular, we evaluated the performance of the system with respect to the ratio of required data processing tasks and network lookup tasks. We further introduced and evaluated an approach to increase the frequency with which linked programs can be executed repeatedly that improves upon a simple sequential re-start of a program.

In summation we can confirm Hypothesis 1, as we have shown how declarative rule-based programs can be utilised to define desired dynamic interactions with Web resources for client applications in such a way that the described interactions can be executed in a highly parallel streaming fashion. Applications built upon such linked programs achieve low runtimes in the face of Web-distributed data resources for the acquisition of information.

The focus of future research work building upon the topics of this chapter can be twofold:

First, a further increase of control over the evaluation of linked programs can result in an additional performance increase. Additional control can relate to more precise restrictions on what links to follow; e.g., one could specify that only resources should be retrieved and included in the processing that are reachable from the initial resources in a defined number of hops. Thus, the processing can be further limited to a subset of resources, which contain the necessary data for the goal of an application. Such control mechanisms would require a precise provenance tracking of the individual statements that are processed, i.e., retaining of information which triple comes from which resource. Provenance tracking for a system that combines reasoning with query answering over interlinked resources presents a challenge, as some statements will be derived from other triples that come from different resources.

Control over the evaluation of linked programs can also relate to a more deliberate scheduling of processing and lookup tasks. E.g., algorithms can be de-

veloped that determine at runtime when the employed queues are filled so that the data processing threads can be saturated and therefore stall further resource lookups. Thus, the size of the employed queues during evaluation could be kept smaller reducing the memory footprint of the evaluation and potentially reducing runtime, as read and write times to the smaller data structures are reduced.

The second aspect of future research work can be a higher level of parallelisation: While we parallelise the evaluation with multiple processes executed simultaneously on different cores of processors in a single machine, a higher level of parallelisation implies the distribution of the evaluation over several machines. Similar to a MapReduce framework [24] several machines can work together in evaluating an evaluation plan. The communication between machines can cause overhead comparable to the overhead caused by the use of QPI when employing multiple processors. Therefore, algorithms have to be developed to share an overall evaluation plan among multiple machines that minimise inter-machine communication. The identification of beneficial strategies to share plans represents a challenge as the processed data is unknown before executing a program.

This chapter was mostly concerned with the realisation of a parallel processing approach for the evaluation of linked programs. Thus the challenge of achieving a low runtimes for Web-based applications was in the focus of this chapter. However, the described processing model adheres to constraints as required in high entropy environments such as the Web. Specifically, the capabilities for the on-the-fly processing of interlinked schemata in conjunction with reasoning facilities for the alignment of resources enable applications to react to dynamically changing Web resources.

We restricted the described approach to linked programs on the retrieval of resources for the extraction of information used by applications. Even though many use cases fall under such scenarios, there are also many tasks a Web application can perform, that require that resources are actively changed by the application. I.e., rather than just retrieving information, an application can also produce new data and change existing resources. We extend linked programs with manipulation capabilities in the following chapter by combining Linked Data with REST.

4

DYNAMIC MANIPULATION OF WEB RESOURCES

4.1 INTRODUCTION

The Web offers more than simple access to data and information as supplied by data providers. Rather a wide variety of possibilities to create and edit data exist, e.g., in the context of user generated content. Existing service offerings on the Web not only go beyond the simple exposure of data by providing facilities for the manipulation of data and offer a broad spectrum of functionalities that can entail real world effects. Examples for such services include online banking, reservation of hotels and flights, ordering of food.

An increasing amount of providers do not only offer such functionality via their homepage (i.e., human-usable interface), but also allow for programmatic access via an Application Programming Interfaces (API). ProgrammableWeb[64], a popular registry for Web-based APIs lists over 13 000 interfaces and describes an exponential increase in the number of available APIs over the last years. Consequently applications like mobile or desktop apps can be built upon the utilisation of these functionalities. An increased value realised by such applications comes from combining data from multiple sources and functionality from multiple providers. The importance of such compositions is also reflected in the constant growth of mashups – small programs that combine multiple web APIs [125].

In addition to the momentum generated by the Linking Open Data community towards opening up public sector and other data [13] as Web resources, there is also a strong movement toward a resource-oriented model of APIs based on Representational State Transfer (REST) [29]. REST gains popularity with an interaction architecture based on the manipulation of resources by fostering a loose coupling between client application and server. A loose coupling between client and server refers to the concept that the client requires only little or no knowledge about the server for a successful interaction. In particular, REST achieves loose coupling by constraining the available operations a client can choose from for an interaction. Furthermore, clients can draw flexibility from links between resources, which allow the clients to discover a priori unknown resources at runtime.

The characteristic of an architecture to be loosely coupled directly relates to the property of an API to present potential clients with little entropy, as the clients can rely on a predefined interaction model. Adaptivity and robustness are direct

[64] http://programmableweb.com; retrieved 2015-03-15

consequences of such an interaction model and are particularly useful for software architectures in distributed data-driven environments such as the Web [82]. However, the resources, which are the target of the interactions, are published using different data formats and non-aligned vocabularies built on heterogeneous schemata. Thus, clients still have to face the entropy from the heterogeneous resource representations, which makes writing programs that integrate offers from multiple providers a tedious task.

In a REST architecture, client and server are supposed to form a contract with content negotiation, not only on the data format but implicitly also on the semantics of the communicated data, i.e., an agreement on how the data have to be interpreted [124]. Since the agreement on the semantics is only implicit, programmers developing client applications have to manually gain a deep understanding of the provided data, often based on natural text descriptions. The combination of REST resources originating from different providers suffers particularly from the necessary manual effort to use and combine them. The reliance on natural language descriptions of APIs has led to mashup designs in which programmers are forced to write glue code with little or no automation and to manually consolidate and integrate the exchanged data.

As described in Chapter 3, Linked Data unifies a standardised interaction model with the possibility to align vocabularies using RDF, RDFS and OWL. Thus, Linked Data specifically mitigates the entropy caused by heterogeneous resources. However, the interactions are currently constrained to simple data retrieval, rather than the active manipulation of resources as intended by REST. Consequently, a combination of Linked Data and REST is mutually beneficial for both approaches. Linked Data offers a uniform data model for REST with self-descriptive resources that can be leveraged to avoid a manual ad-hoc development of Web-based applications. Following the motivation to look beyond the exposure of fixed datasets, the extension of Linked Data with REST technologies has been explored [10, 127] and led in particular to the establishment of the Linked Data Platform (LDP)[65] W3C working group. LDP devises a recommendation [101] with a set of rules regarding the use of HTTP for accessing, updating, creating and deleting Linked Data resources, which are denoted with *Read/Write Linked Data*. Several existing approaches recognise the value of combining REST services and Linked Data [61, 102, 120, 105].

In this chapter we study synergies and discrepancies in the combination of Linked Data and REST, based on our work in [108]. Specifically, we propose Linked Data-Fu (LD-Fu), a data- and resource-driven programming approach for the development of applications built on Semantic Web resources with a declarative rule language. The lightweight rule language of LD-Fu is an extension of linked programs introduced in Chapter 3.

The goal of our work is to reduce the tedious effort to develop applications in a high-entropy environment such as the Web, by providing the means to specify application logic including the active change and creation of Web resources. While preserving the possibility to realise applications with short response time, such declarative specifications provide a modular way of composing the functionality of multiple APIs and preserve loose coupling by

[65] http://www.w3.org/2012/ldp/charter

- leveraging links between resources provided by Linked Data, and
- specifying desired interactions dependent on resource states, which is enabled by a uniform state description format, i.e., RDF,
- specifying the required reasoning constructs to align resources.

LD-Fu programs, can be evaluated with the same parallel push-based execution model as detailed in Chapter 3. Consequently, we retain similar performance characteristics, that allow to realise applications with low runtime requirements, also for Web applications that manipulate resources. While there has been recent work on extending the Map/Reduce model for data-driven processing [54, 6], these approaches are geared towards deployment in data centers. In contrast, our approach operates on the networked open Web.

✎ **Example 18**

ACME aims at extending their social media activities to a broader range of dissemination channels (for further information on multi-channel communication see [16]). Acme's marketing department observes that while the number of potential channels is constantly increasing, the channels can be broadly categorised into micro blog services and social networks. Information about upcoming events, special offers, and other news should be disseminated in the following ways:

- Posts on the company's micro blogs
- Messages to social network users who are followers of the company.

Both ways to disseminate ACME's news require the creation and update of resources on the Web. We assume that the dissemination channels offer Linked APIs, i.e., resources are exposed that offer read/write Linked Data functionality.

The marketing department orders a system from Acme's IT that manages the dissemination channels and automatically disseminates a post to all available channels either as a micro blog entry or as a personal message. Initially the micro blog service *MB* and the social network *SNA* have to be supported. Marketing will supply their posts in an Acme-specific vocabulary as so-called opInfoItems.

After a while, the marketing department decides to add the new social network *SNB* as a dissemination channel, which requires two steps:

- The IT department extends the dissemination system to support the interface of SNB; and
- The marketing department adds Acme's identity in SNB to the dissemination channels.

Marketing wants to promote sales, where special offers for concert tickets are created. The sales will be implemented by posting InfoItems with the offer to the available dissemination channels.

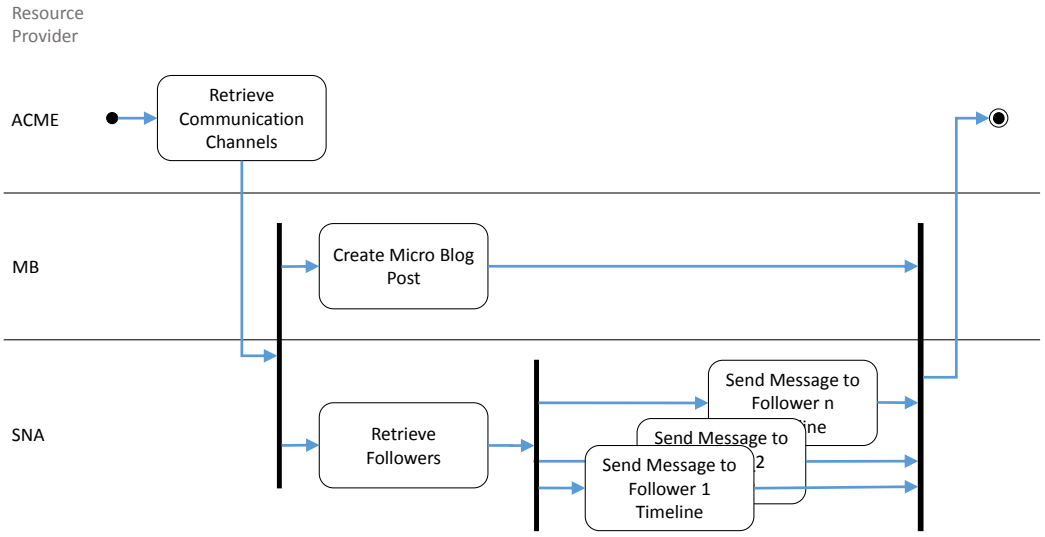


Figure 19: UML Activity Diagram illustrating the dissemination of news.

Table 7: URI prefixes used for social networks and micro blog

Prefix	IRI
sna:	http://sna.example.org/lapi/
snb:	http://snb.example.org/rest/
mb:	http://mb.example.org/interface/

Throughout this chapter, we will illustrate technical contributions by realising bits and pieces of the proposed scenario. When modeling services and interactions, we will use a number of URI prefixes for the fictitous microblog and social networks as listed in Table 7.

Figure 19 illustrates the activities related with the dissemination of news.

4.1.1 *Challenges*

We address the following challenges in the context of runtime requirements and entropy (see Section 1.1.1):

- The use of Linked Data provides REST with a uniform way to represent resources. However, both REST and Linked Data imply their own interaction model. Although related, the interaction models exhibit differences, which relate to the focus of both approaches on data provisioning and resource manipulation respectively. E.g., Linked Data imposes a distinction between a resource representing a real world object and the resource that is the document describing the object (i.e., information resource) [29]. REST does not know such a distinction and generally equates the object and the data describing it. Such differences have to be resolved in a way to leverage the advantages of of both approaches.

- REST is a general software architecture. The specification of HTTP constitutes the definition of the implementation of REST on the Web. To achieve the desired effect of reducing the entropy of APIs and in turn the manual effort for the creation of applications, we must define a rigorous formal model for the interaction with Linked Data-based REST APIs. The formal model is required, so that client applications can rely on a specific behaviour of an API. Consequently the formal model has to be general enough to cover many use-cases.

- A program defining application logic must combine facilities for dynamic link traversal with the possibility to specify the kind of intended interaction as well as the intended result of a resource manipulation. Specifically, it has to be possible to describe how and under what conditions resources, which are discovered at runtime, are supposed to be changed or deleted. Furthermore, it has to be possible to describe under what condition resources have to be created. The instructions for the manipulations, i.e., the messages sent to resources, have to respect the vocabulary that is accepted by a given API. The alignment of manipulation instructions with the rest of the data processing can build upon predefined reasoning constructs. Specifications for manipulations have to be precise and fine-grained to prevent an application from doing unnecessary work. However, it must also be possible to define manipulations that affect a large set of resources at once to keep the program specification simple.

- The execution of intended resource manipulations has to be integrated into the scheduling push-based execution model to achieve short runtimes. However, the manipulation of Web resources can result in race conditions, which are not inherently preventable in a REST architecture, especially if manipulations are carried out in parallel. E.g., the effect of two operations to overwrite the same resource with different data depends on the order in which the operations are completed. Consequently, the effected result of a program, i.e., the state of the Web resources after execution of a program is not necessarily deterministically defined.

4.1.2 *Contributions*

Our contributions are as follows:

- We identify discrepancies and synergies between Linked Data and REST. In particular, we describe how the differences can be resolved and how both approaches can be combined to *Linked APIs*. Linked APIs overcome drawbacks and realise advantageous of both approaches. The described combination defines a general architecture for APIs that remains compatible with the LDP recommendation. We further introduce methods to create Linked APIs by wrapping existing REST APIs on the Web. (Section 4.2)

- We introduce a model for REST APIs based on state transition systems as formal grounding for the manipulation of resources exposed in Linked APIs. Specifically we define the combined representation of Web resources

as states, which allows to interpret manipulation upon the resources as transitions between states. Thus, requests to manipulate resources can be interpreted to trigger transitions in a formal state transition model. (Section 4.3)

- We extend our previous definition of rule-based programs to allow the specification of resource manipulations. Specifically LD-Fu programs are a superset of linked programs, which allow for rules that cause state transitions in the described model for Linked APIs. The manipulated resources as well as the messages to trigger state changes can be dynamically derived from the processed data, i.e., from retrieved resources. Thus, our approach allows for a definition of intended resource changes conditioned to the current state of resources leveraging reasoning capabilities to align heterogeneous vocabularies. The evaluation of LD-Fu programs can be done with a parallel push-based execution model, to achieve low runtime requirements similar to linked programs. (Section 4.4)
- We characterise the potentially non-deterministic behaviour of LD-Fu programs and introduce strategies to avoid or mitigate undesired effects. (Section 4.4.2) We detail the behaviour of the repeated evaluation of programs with resource manipulations and show how an iterative execution can be employed to achieve the goals of an application. (Section 4.4.3)

We describe experiments in Section 4.5, cover related work in Section 4.6, and conclude the chapter with Section 4.7.

4.2 COMBINING LINKED DATA AND REST

Both Linked Data and REST are concerned with the provisioning of Web resources as well as the interaction of clients with such resources. Both approaches overlap with respect to certain aspects of the interaction, but also cover topics that are not addressed in the respective other approach. A combination of Linked Data and REST extends both approaches in the following way:

- Linked Data receives capabilities to manipulate Web resources beyond the the consumption of data, i.e., resources can be changed rather than just retrieved. With a focus on the aspect to allow for resource manipulation, the combination is denoted as read/write Linked Data [101, 10]
- REST receives a uniform data-model for self-descriptive resources, such that clients can interpret the resources according to Linked Data principles. With a focus on the aspect to leverage the LD data model, the combination is denoted as Linked API [107].

Both read/write Linked Data and Linked APIs can be used synonymously, as they describe the same combined approach to resource interaction. In particular, read/write Linked Data stresses the extension of Linked Data with capabilities of REST, and Linked APIs stress the extension of REST with capabilities of Linked Data. However, both extensions converge to the same approach.

Linked Data and REST can be characterized with a set of principles [13, 94] (see Sections 2.2 and 2.3). In the following we describe differences and synergies of Linked Data and REST, thus characterizing how both approaches can be used together.

4.2.1 *URI-identified Resources*

Linked Data is about the provision of information as resources identified by URIs, which is encoded in the first Linked Data principle. In the same way REST is concerned with uniquely identifiable resources.

Linked Data	REST
Use URIs as names for "things".	*Use URI-identified resources..*

Linked Data specifically addresses Web resources, therefore the URIs are specifically defined to be built upon HTTP. REST is a general software architecture [29] that not necessarily uses Web technologies, therefore the URIs can adhere to an arbitrary protocol. However, the Web is the main application domain of REST. At least in the context of the interaction with Web resources we can assume that both approaches use HTTP URIs to identify resources.

However, the understanding of what a resource is differs slightly. While anything can be identified by a URI, REST does not distinguish between the document describing an entity and the entity itself, in the same way it is done in Linked Data. In REST a lookup of a client on a URI that identifies a real world object can directly return data about the object, while Linked Data implies that the client is referred to the URI identifying the document containing the data. The distinction between the information resource of an entity and the resource representing the entity itself does not contradict the principles of REST, as the documents can just be considered resources in the sense of REST. Information resources add the possibility to make meta-statements about the data describing entities. We assume that Linked APIs generally use Linked Data resources, i.e., resources that adhere to the Linked Data principles, including the distinction for information resources.

4.2.2 *Interaction Methods*

The interaction with a Web resource is encoded in the second and third Linked Data principles as lookups. In contrast REST just generally defines the use of methods.

The idea behind REST is that applications use functionalities provided on the Web via APIs that are not based on the call of API-specific operations or procedures, but rather on the direct interaction with exposed resources. While Linked Data only talks about the lookup of URIs, REST just implies a constrained set of operations for interactions. Again, in the context of Web resources we assume

Linked Data	REST
Use HTTP URIs so that people can lookup the names. *When someone looks up a URI, provide information, using the standards.*	*Use a constrained set of operations..*

the HTTP methods $M = \{\mathsf{GET}, \mathsf{PUT}, \mathsf{POST}, \mathsf{DELETE}\}$[66] to be the operations in a REST architecture. Thus, REST goes beyond the principles of Linked Data by allowing for unsafe methods, which we adopt for the combination of both approaches. The restriction of the allowed operations helps in the reduction of entropy, as the clients can rely on the predefined implications of a small set of methods.

The representation of a resource (i.e., the data returned from a lookup) details the current state of the resource, therefore the representation can dynamically change. A manipulation of the state representation with REST sometimes implies that the represented resource is manipulated accordingly.

The distinction between resources and their information resources of Linked Data can help to clarify if a manipulation only refers to data change or has an actual real world effect. Specifically, if an entity represented by a resource r cannot be changed by manipulating the data representation, a server can forbid requests with unsafe methods to be directed at the resource. However, manipulating the data representation can still be possible, e.g., for the purpose of complementing information. A request can be directed directly at the information resource u with $r \xrightarrow{i} T^u$.

✎ **Example 19**

The resource `acme:order37` represents an ordering of tickets for a concert. A user might overwrite this resource with a request to change the amount of ordered tickets for the concert $(\mathsf{PUT}, \texttt{acme:order37}, D^{req})$. As a response the server refers the client (`303 SEE OTHER`) to the information resource `data:order567`, which the client can change with his request. Consequently the actual order (i.e., the real world object) has changed.

The resource `sna:user84` represents a user of a social network SNA. If an attempt is made to overwrite the resource $(\mathsf{PUT}, \texttt{sna:user37}, D^{req})$, the server can deny the request (`405 Method Not Allowed`), as the user itself cannot be changed.

[66] For brevity we focus on the non-safe methods, as well as GET as defined in RFC 7230. Other HTTP methods can be added in a straight forward manner. See https://tools.ietf.org/html/rfc7230; retrieved 2015-04-10.

> However, to manipulate the information resource `sna:userProfile84` of the user with `sna:user84` $\xrightarrow{i}$ $T^{\texttt{sna:userProfile84}}$ to change a wrongly entered birthday remains possible.

It has to be noted, that if a server relies on hashtags to differentiate information resources, the server exhibits no direct control over the referral: Every request to a resource is automatically referred to an information resource. Thus, the approach as described in Example 19 cannot be employed to clarify, if the application of an unsafe method actually influences the represented object.

Furthermore, a client executing a request on an information resource directly, cannot in any case predict if the effect of the request extends to a real entity. In fact, in the absence of other information, a client should always expect that an information resource reflects the current state of an entity correctly, and consequently that a manipulation of the information resource extends to the entity.

REST does not impose a specific data format or model for the representation of a resource. In an interaction of an application with a resource, a contract has to be established between client and server in a process called content negotiation (conneg). This contract represents an agreement that the server provides the representation of the resource in a format that is understood by the client. If server and client cannot identify a common format, an interaction cannot take place.

In different application scenarios REST assumes vendor specific content types for the contract between client and server to convey the meaning of the communicated data. The idea behind vendor specific content types is, that service providers can reuse content types and client applications can make use of specific content type processors for the individual content types. In practice however, we see that many REST APIs on the Web simply make use of standard non-specific content types, e.g., *text/xml* or *application/json* [68]. Further, API providers sometimes offer a programming library or a Software Development Kit (SDK) to support the use of their exposed API in client applications (e.g., see the Facebook API SDKs[67]). Such SDKs increase the coupling between client and API and require developers to write glue-code if an application is supposed to combine the functionality of multiple providers. Also API SDKs are mostly only applicable for the APIs of one provider, i.e., SDKs cannot be re-used for other APIs as it is intended with content type processors.

In contrast, Linked Data imposes the use of a standard semantic representation format, i.e., RDF. Thus, clients can leverage Web-accessible schemata for the interaction with Linked Data resources and in conjunction with reasoning capabilities integrate resources from different providers. The interpretation of different resources can be done in a uniform way. Therefore, if the employed schemata are interlinked, the entropy with respect to the representation of resources refers mostly on the translation of different RDF serialisation formats.

The effect of a request $(m, r, D^{req}) : m \in \{PUT, POST, DELETE\}$ with non-safe method on the state of an addressed resource r can depend on the input data D^{req}. The communicated input data can be subject to requirements that need to be described to allow an automated interaction, e.g., the input data can impose

[67] `https://developers.facebook.com/docs/apis-and-sdks`; retrieved 2015-04-10.

the use of a specific vocabulary. The dependency between communicated input and the resulting state of resources also needs to be described. We cover such description mechanisms in Chapter 5.

The state of a Linked Data resource is expressed with RDF, which is returned in the response of a lookup. Therefore, it is sensible to serialise the input data D^{req} in RDF as well, i.e., data that is submitted to resources to manipulate their state.

4.2.3 *Hypermedia Links*

Both Linked Data and REST agree on the use of links, however, their principles have a different point of view.

Linked Data	REST
Include links to other URIs in the descriptions to allow people to discover more "things".	*Apply hypermedia controls as engine of application state.*

Linked data mandates links between resources to allow for the discovery of further relevant information. Where Linked Data is only concerned with the retrieval of information, REST bases the manipulation of resources also on the discovery of resources via links.

The flexibility of REST results from the idea that client applications do not have to know about all necessary resources. The retrievable representations of some known resources contain links to other resources, that the client can discover during runtime. Clients can use such discovered resources not only to acquire further information, but also to perform manipulations upon them.

Understanding the goal of an application not only to be the acquisition of information by data processing, but also the modification of resources into an intended state, the links reflect possible transitions of resource states towards the goal of an application. Consequently, links between resources are referred to as hypermedia controls. The point of view to interpret the links as the elements driving an application towards its goal is called Hypermedia as the engine of application state (HATEOAS) [29].

The development of applications in a REST framework is especially challenging, since the links between resources and the resource states can only be determined during runtime, However, programmers have to specify their desired interactions including intended changes at design time. To respect the link architecture allows client application to not rely on hard-coded manipulations, but dynamically address resources as required. Specifically, a client can retrieve a set of resources, and manipulate them as a consequence of their current state. The reaction on state changes becomes especially important in a distributed programming environment, since a client cannot ex ante predict the influence of other clients on the resources, i.e., REST does not allow a client to make assumptions on resource states. The dynamic reaction to state changes of resources is in

particular possible, because interactions with resources are stateless, i.e., requests and responses are self contained and can be executed independently from a priori knowledge about the state of the addressed resource.

✎ **Example 20**

ACME's IT creates the resource `acme:acme` representing the company ACME itself. A lookup $(\mathsf{GET}, \texttt{acme:acme}, \emptyset)$ redirects to the information resource `data:acme`, which in turn leads to a response $(200, \mathrm{D}^{res})$ with the following initial RDF graph D^{res} as payload:

```
acme:acme rdf:type p:Company .
```

The marketing department updates the `acme:acme` resource with the dissemination channels SNA and MB with a request $(\mathsf{PUT}, \texttt{data:acme}, \mathrm{D}^{req})$ with the following graph D^{req} as input data:

```
acme:acme rdf:type p:Company .
acme:acme p:dissChannel sna:acmeID, mb:acme .
sna:acmeID rdf:type p:SocialNetworkID .
mb:acme  rdf:type p:MicroBlogTimeline .
```

A subsequent lookup on `acme:acme` would result in exactly the description that marketing supplied with their PUT request.

A lookup $(\mathsf{GET}, \texttt{sna:acmeID}, \emptyset)$ on ACME's identifier in the social network SNA, would result in a description of ACME in SNA's vocabulary, including its fans:

```
sna:acmeID  rdf:type sna:CommercialOrganisation .
sna:acmeID sna:founded "03/07/1984" .
sna:acmeID sna:fan sna:userA, sna:UserB, ... .
```

The resources representing users in the SNA network provide functionality to send messages to the corresponding users. A POST can be employed to send a message to a user resource (e.g., to `sna:userA`). A request $(\mathsf{POST}, \texttt{sna:userA}, \mathrm{D}^{req})$ details the message to be sent, the sender, as well as relevant links for the message in the payload D^{req}:

```
[] rdf:type sna:Message ;
   sna:sender acme:acme .
   sioc:content "Upcoming concert of Metallica at ..." .
   rdfs:seeAlso acme:concert123 .
```

Acme's timeline `mb:acme` on the micro blogging service MB also supports the POST operation: Similarly a request $(\mathsf{POST}, \texttt{mb:acme}, \mathrm{D}^{req})$ can be used to create a new resource, which is a distinct entry in the timeline containing a message and relevant links:

```
[] rdf:type sioc:Post ;
   sioc:content "Upcoming concert of Metallica at ..." .
   rdfs:seeAlso acme:concert123 .
```

Applying a DELETE on a blog post, e.g., one that advertises an expired sale, does not require input; its effect is inherently defined by the method: the entry is erased.

Definition 13: Linked API

A Linked API is a set of Web resources R, such that

- every resource is identified with an HTTP URI, $\forall r \in R : r \in \mathcal{U}$;
- every resource allows successful requests
  - with RDF graphs as input and output payload, and
  - with HTTP methods

  either directly or the resource refers to an information resource of the linked API that allows successful requests, i.e., $\forall r \in R \; \exists m \in M : \beth(m, r, D^{req}) = (2xx, D^{res}) \vee \beth(m, u, D^{req}) = (2xx, D^{res})$ with $r \xrightarrow{i} T^u$, $u \in R$, $M = \{GET, PUT, POST, DELETE\}$, and D^{req}, D^{res} are RDF graphs; Also empty graphs are allowed as input and output payload $D^{req} = \emptyset$ or $D^{res} = \emptyset$.
- some resources contain links to other resources, $\exists S \subset R$ such that $\forall r \in S \; \exists u \in \mathcal{U} : r \xrightarrow{l} u$.

4.2.4 *Wrapping APIs*

It should be noted that due to the Definition 13, every existing set of Linked Data resources is a Linked API. The constrained that client applications can address resources with requests using any HTTP methods, is the only aspect of the definition that goes beyond the Linked Data principles. However, we cannot enforce that a resource is allowing methods that are different from GET. Indeed, if a provider intends an API to just provide information to client applications, without any interactivity, i.e., the data should not be changed, extended or deleted by the client, the provider only needs to allow lookups.

However, there is a large amount of APIs on the Web that allow for dynamic manipulations of resources. But only few of them adhere at the same time to Linked Data principles, i.e., are Linked APIs[68].

Existing REST APIs can be easily extended to be Linked APIs, thus allowing clients for an easy integration of the functionality offered by various providers. A provider can modify the existing conneg of an API to allow clients to request resources represented in an RDF serialisation. Existing hypermedia controls in the API become links between the Linked Data resources. However, for competitive economic reasons a provider might not be interested to foster the composition of APIs from other providers with their own APIs. Especially providing links to

[68] See http://lapis.planet-data.eu/, retrieved 2015-04-10, for a small catalogue of existing Linked APIs.

external resources offered by other providers might be seen as problematic, as the external resource cannot be directly controlled. A link to an external resource potentially leads customers away from the provider.

APIs can also be extended to offer Linked Data functionality without intervention of the original provider by a process called *wrapping*. Wrapping refers to the concept of mapping elements of input and output messages of a service to ontological concepts, thus enriching the messages with semantic constructs for the interpretation by a client. The annotation of messages of operation-oriented services is described in the W3C recommendation for Semantic annotations for WSDL (SAWSDL) [59, 25]; an approach for the annotation of REST APIs is Semantic annotations for REST (SA-REST) [64]. Here, we just discuss the general concept and feasibility of wrapping REST-based APIs [61].

In particular, a wrapper is a software component that intervenes in the communication between a client application and an API. In the case of linked APIs a client sends and receives RDF graphs as payload of requests and responses via the wrapper, which triggers a subsequent communication with the actual API. Thus, the wrapper acts as mediator between client and API by exposing Linked Data resources linked to other Web resources of the API in two ways:

LOWERING The client can formulate a request with an RDF graph D^{req} as payload to a resource of the wrapper. The wrapper translates the payload according to a predefined mapping into a data model and format understood by the actual API. The wrapper sends a request with the translated payload to the API.

LIFTING The wrapper receives responses from the API and translates the payload into and RDF graph D^{res} according to a predefined mapping. The wrapper sends a response with the translated payload to the client.

✎ **Example 21**

ACME's marketing recommends to include the small social network SNC as a dissemination channel. However, SNC does not offer a Linked API, but a REST API which allows the interaction with resources as JSON objects.

ACME's IT creates an account on SNC. As ACME's analysists do not expect that SNC will offer a Linked API in the near future, ACME's IT develops and deploys a wrapper. The wrapper contains mappings from JSON keys to RDF properties of ACME's and other vocabularies as follows:

JSON key	RDF property
`"founded"`	`p:founded`
`"fans"`	`p:fan`
`"from"`	`p:sender`
`"message"`	`sioc:content`
`"link"`	`rdfs:seeAlso`

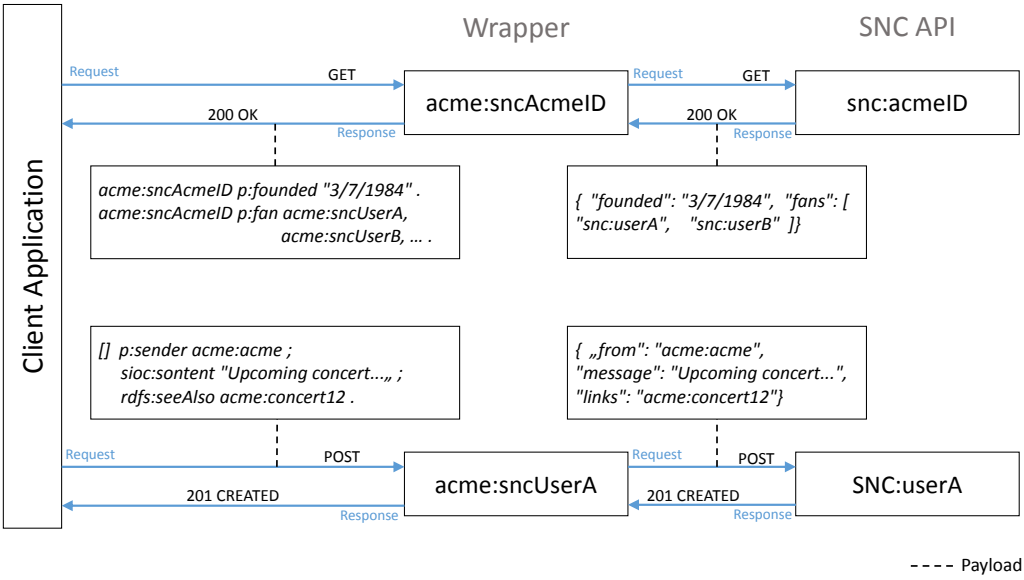


Figure 20: Illustration of a wrapper mediating between a client application and a social network API.

> The wrapper exposes Linked Data resources corresponding to the resources of SNC's API. Clients can interact with the resources of the wrapper, which uses the mappings to consruct JSON objects from RDF graphs and vice versa for underlying interactions with SNC's API. Figure 20 shows a lookup on ACME's account and the submission of a message to a user on SNC via the wrapper.

Multiple resources from various providers that do not offer Linked APIs can be combined via wrappers leveraging schemata and reasoning as e.g., described in Chapter 3. Thus, wrappers provide a homogeneous interface for heterogeneous data sources. For an approach to combine various data sources in a wrapper-mediator architecture see [126].

Even though several wrappers could use mappings to the same vocabularies, the wrappers have to be considered specific for individual APIs not for an application; the integration can be done by the client application. Combining APIs via wrappers is scalable, as only one wrapper needs to be provided for a data source, rather than having a mediator for every pair of sources to be combined. In particular, while a wrapper can be an integral part of the application using an API, the wrapper can also be independently deployed to be accessible on the Web[69]. Thus, further improving the scalability of the employment of wrappers, as multiple applications can make use of the same wrapper to integrate the functionality of the underlying API.

It is noteworthy that the effort to create wrappers for an APIs, can be expected to be similar to directly integrating the APIs in an application by writing custom glue code, because all elements of messages to and from the resources of the API need to be mapped. Furthermore, sometimes elements can not be directly

[69]The tunneling of functionality via a wrapper to an arbitrary set of users and clients can be in conflict with the terms and conditions of the original provider.

mapped and require additional transformations, e.g., the serialisation format of a date must be changed to adhere to the ontology standard. However, as a wrapper needs only to be created once, the wrapper subsequently reduces the effort to integrate the corresponding API in other applications, potentially also by third parties.

Additionally, the process to create a wrapper can be automated to a large extent. Only the concrete association of message elements to ontological concepts requires manual intervention, as it adds additional information to the wrapper. This additional information reduces the entropy for the interaction with the API to effectively enable clients applications to leverage the schemata for the alignment with other resources. There are data integration tools available that support users in the mapping process by providing graphical interfaces, learning mechanisms and an automated export of transformation rules derived from the mappings [47, 116]. An example of such a tool is Karma[70]. Experiments have shown that Karma can be directly deployed on the Web, where it can make example calls to an API that is to be wrapped. Users can provide the mappings for the example calls and immediately deploy a Web-accessible wrapper, thus cutting the time required to create a wrapper down by orders of magnitude compared to a manual wrapper implementation for Linked APIs [52].

4.3 LINKED API INTERACTION MODEL

A Linked API can be identified with the resources it exposes. An interaction within a REST architecture is based on the manipulation of the states of these exposed resources. While we often denote a set of resources offered by the same provider as a specific Linked API, there are no constraints on the origin of the resources. Intuitively, a client can interact with a set of resources, which are offered from a variety of providers in the same way as it would with resources from a single providers. Indeed, due to the homogeneous interface a client might not even be able to distinguish between resources from different providers, except for the different top level domains of the URIs identifying the resources.

To allow client applications to dynamically perform interactions, the client has to be able to rely on a uniform and defined behaviour of the resources. We develop a model, that allows to formalise the functionalities exposed by a Linked API. A formal service model serves as rigorous specification of how the use of individual HTTP methods influences resource states and how these state changes are conveyed to interacting clients. We derive the model from the RFC specification for HTTP[71] based on the requests and responses to Linked Data resources. The specification allows for several optional features and allows APIs a certain variety in the implementation, resulting in increased entropy. Additionally in a real setting on the Web, providers might not completely adhere to the specification. Consequently, we focus on what we consider the bare minimum of a convention on the behaviour of Linked APIs, which client applications have to

[70] http://www.isi.edu/integration/karma/; retrieved 2015-04-10.
[71] https://tools.ietf.org/html/rfc7230; retrieved 2015-04-10.

rely on[72]. Client applications that interact with resources that offer more features as covered by our model, can choose to ignore these features. E.g., if a response contains information in the payload, where our model defines responses with an empty payload, the client can just discard the retrieved information.

We define the interaction model of Linked APIs as a Linked Data State Transition System (LDSTS) similar to a state machine as defined by Lee and Varaiya [65] and to Mealy machines [72], where a state transition is dependent on the previous state and an input alphabet. The behavior of the clients themselves is not in the scope of this model, it rather formalises all possible interaction paths of a client with a set of resources.

➳ Definition 14: Linked Data State Transition System

A LDSTS is defined as a 5-tuple
$\text{LDSTS} = (R, \Sigma, I, O, \delta)$ with:

- A set of URI-identified resources $R = \{r_1, r_2, ...\}$.
- A set of states $\Sigma = \{\sigma_1, ..., \sigma_m\}$. Each state $\sigma_k \in \Sigma$ of the LDSTS is defined as the union of the states of all resources: $\sigma_k = \bigcup_{r_i \in R} \overline{r_i^k}$. The state of a single resource $r_i \in R$ in a state σ_k is given by its RDF representation, i.e., a graph $\overline{r_i^k} \subset G$, where G is the set of all possible RDF triples.
- An input alphabet $I = \{(m, r, D^{req}) : M \times R \times 2^G\}$, which is the set of possible requests, where
  $M = \{\text{GET}, \text{DELETE}, \text{PUT}, \text{POST}\}$ is the set of the supported HTTP methods[a] and $D^{req} \in 2^G$ an RDF graph as input payload.
- An output alphabet $O = \{(c, D^{res}) : C \times 2^G\}$, which is the set of possible responses, where C is the set of all HTTP status codes and $D^{res} \in 2^G$ an RDF graph as output payload..
- An update function $\delta : \Sigma \times I \to \Sigma \times O$ that returns for a given state and input the resulting state and the output. We decompose δ into a state change function $\delta^s : \Sigma \times I \to \Sigma$ and an output function $\delta^o : \Sigma \times I \to O$, such that $\delta(\sigma, i) = (\delta^s(\sigma, i), \delta^o(\sigma, i))$. We define the state change function as

$$\delta^s(\sigma_k, (m, r_i, D^{req})) = \begin{cases} \sigma_k, & \text{if } m = \text{GET} \\ \sigma_k \setminus \{\overline{r_i^k}\}, & \text{if } m = \text{DELETE} \\ (\sigma_k \setminus \{\overline{r_i^k}\}) \cup D^{req}, & \text{if } m = \text{PUT} \\ \text{post}_i^s(\sigma_k, D^{req}), & \text{if } m = \text{POST}, \end{cases}$$

[72] An API adhering to the RFC7239 et seq. and the LDP is compatible with our interaction model, such that a client relying on our interaction model can interact with the API.

where the function $post_i$ encapsulates the resource specific behaviour of a POST request. We define the output function as

$$\delta^o(\sigma_k, (m, r_i, D^{req})) = \begin{cases} (2xx\,, \overline{r_i^k}), & \text{if } m = \text{GET} \\ (2xx\,, \emptyset), & \text{if } m = \text{DELETE} \\ (2xx\,, \emptyset), & \text{if } m = \text{PUT} \\ (2xx\,, post_i^o(\sigma_k, D^{req}), & \text{if } m = \text{POST} \end{cases}$$

[a] For brevity we focus here on the four most important methods, i.e., unsafe operations and GET.

For brevity we assume all resources to be information resources. The model can, however, be trivially expanded to general Linked Data resources, where for every request on a resource (m, r, D^{req}), the response refers to the corresponding information resource u with $r \xrightarrow{i} T^u$. Furthermore, we only describe successful requests and omit requests that result in responses with error codes 4xx or 5xx. Again, the model can be trivially expanded for unsuccessful requests, as they do not result in a state change.

Resources do not necessarily allow the use of all HTTP methods. Note that all state change functions are defined for every resource, i.e., every resource can be addressed with all methods: If a resource does not allow for the application of a specific method, the response contains an error code and the state change function describes a self-transition.

A client interacting with a Linked API modeled by an LDSTS $= \{R, \Sigma, I, O, \delta\}$ creates an input $i = (m, r_i, D^{req})$ (i.e., a request) for LDSTS by invoking the HTTP method m on the resource r_i and passing the potentially empty RDF graph g in the request body. Depending on the current state σ_k the following happens:

1. A transition into the state $\delta^s(\sigma_k, (m, r_i, D^{req}))$.
2. The client gets an HTTP response with the HTTP code c and the RDF graph D^{res} in the body, where $(c, D^{res}) = \delta^o(\sigma_k, (m, r_i, D^{req}))$.

Safe methods (i.e., GET) that do not change any resource states, describe self-transitions, i.e., transitions that start and end in the same state. The output function for a GET returns the current state of the addressed resource $\overline{r_i^k}$ as RDF graph, which is a part of the current state as $\overline{r_i^k} \subset \sigma_k$.

Unsafe methods (i.e., PUT, POST, and DELETE) potentially cause a state transition. The state change resulting from a DELETE is always that the addressed resource does not longer exist in the next state. A PUT results in adding the payload D^{req} to the next state as resource r_i; if r_i already existed in the current state it is removed. The precise state change effect of a POST depends on the resource r_i it is applied to and is defined by a function $post_i^s : \Sigma \times 2^G \to \Sigma$. Following to the RFC specification a POST allows various effects:

- The creation of a new resource defined by the payload D^{req} with URI chosen by the server.

- The extension of the addressed resource r_i with the triples in D^{req}, similar to appending data to a file.
- The submission of the the payload D^{req} to a data-handling process. Such a data handling process can have no effect on existing resources, create new resources or modify resources and is therefore similar to a general remote procedure call (RPC)[73].

The output function for the unsafe methods PUT and DELETE generally only considers an empty payload with a status code. The output for a POST request depends on the various effects a POST request can have. Thus, output for a POST request is also defined by a function $post_i^o : \Sigma \times 2^G \rightarrow G$, which might detail the effected state change or provide the output of the data-handling process triggered by the POST request.

We do not require an application to react on the payload in the response to an unsafe method, as the application can apply GET requests to retrieve the state of resources (see Section 4.4.3).

It is noteworthy that all unsafe methods can have cascaded effects, i.e., resources might be changed that are not specified as target r_i in the request. E.g., creating a resource with PUT might create a link in a collection resource. For unsafe methods such cascaded effects are not in the scope of the model. A client application can only rely on a change of the addressed resource. However, the precisely effected change is only defined for the idempotent methods (i.e., PUT and DELETE). For the remainder of this chapter we do not consider cascading effects explicitly. However, a client application cannot distinguish anyway between a resource that changed on its own, due to interaction of another client, or a cascading effect of a request executed by the application itself. Thus, we cover cascading effects insofar we consider Web resource to change dynamically.

The defined model serves as formal grounding of the execution language described in Section 4.4.

> ✎ **Example 22**
>
> Figure 21 illustrates a state transition in an LDSTS, where an entry is created on ACME's time line on MB. State σ_1 contains the RDF graphs of the the time line and messages on MB.
>
> After executing a request $(\text{POST}, \texttt{mb:acme}, D^{req})$, where the payload D^{req} defines a new entry, the state transitions into σ_2. In σ_2 the new message exists.

4.4 THE LINKED DATA-FU LANGUAGE

In this section, we describe how linked programs (see Definition 11) can be extended to allow for the manipulation of resources. Specifically, we introduce Linked Data-Fu (LD-Fu)[74], as execution language to instantiate a concrete interac-

[73] https://www.ietf.org/rfc/rfc1057.txt; retrieved 2015-4-10.

[74] We use the name *Linked Data-Fu* in adaption of the term *google-fu*, which adopts the suffix *Fu* from Kung Fu, implying great skill or mastery. Thus Data-Fu hints at the mastery of data interaction that can be achieved with the language.

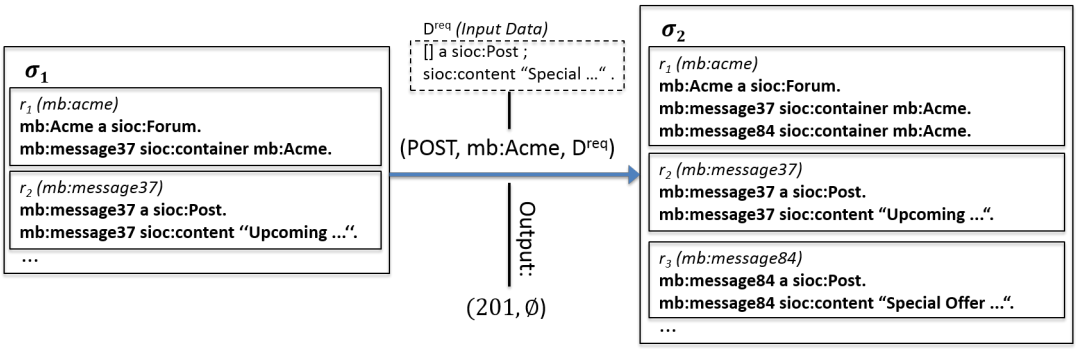


Figure 21: State transition of a LDSTS, with excerpts of two states.

tion between a client and resources, which preserves the adaptability, robustness and flexibility of REST.

In a resource-driven environment, applications retrieve and manipulate resources exposed on the Web. The manipulation has to be dynamic in two aspects:

- The resources that are to be changed might not be known before runtime, i.e., the resources have to be discovered via link traversal.
- Manipulations might be conditioned on specific circumstances under which they are to be executed.

Both, links and the current circumstances are reflected in the current state of an LDSTS. A linked program retrieves resources as specified (request rules) and materialises implicit information (deduction rules) from the retrieved resources including a combination with local information (initial graph and requests). Thus, a linked program acquires a snapshot impression of the current state of the LDSTS, which is constituted by the resources $R_{\mathcal{P}}$ that are retrieved by the program. This snapshot includes initial resources as well as resources retrieved via request rules. Further let $\text{LDSTS}_{\mathcal{P}} = (R_{\mathcal{P}}, \Sigma_{\mathcal{P}}, I, O, \delta)$ be the LDSTS that is constituted by the resources retrieved by a linked program $\mathcal{P}$.

The result graph $G_{\mathcal{P}}$ of a linked program $\mathcal{P}$ represents the entirety of information available to an application about the current state σ from the execution of $\mathcal{P}$. Assuming that the resources $R_{\mathcal{P}}$ are fixed and stable in the current state $\sigma \in \Sigma_{\mathcal{P}}$ of $\text{LDSTS}_{\mathcal{P}}$, it holds true that $\sigma \subseteq G_{\mathcal{P}}$. As per definition every resource of $\text{LDSTS}_{\mathcal{P}}$ is retrieved by the the linked program $\mathcal{P}$, the graph representation of every resource in the current state is included in the result graph $G_{\mathcal{P}}$. We denote $G_{\mathcal{P}}^{\sigma}$ as the result graph of linked program $\mathcal{P}$ executed at state σ.

However, since resources can potentially change on their own or can be accessed and changed by other clients, the result graph of a linked program might not accurately represent a distinct state of an LDSTS. Specifically, resources can change while the linked program is still running, thus the result graph might contain information from different states of $\text{LDSTS}_{\mathcal{P}}$. Furthermore, resources might change after a linked program reaches its fixpoint, thus the state reflected in the result graph is outdated. Ultimately, in a dynamic Web environment it cannot be avoided that an application built upon remote resources bases its decisions on only approximate information.

We enable applications to man ipulate resources based on their available information about the current state of the resources, i.e., the application reacts dynamically on the state of the resources as derived by a linked program. Consequently, there is a dependency between the state transitions in an LDSTS invoked by an application and the current resource states. The dependency between the invoked state transitions (i.e., applied HTTP methods) and the states of resources is that

1. links to the resources that are to be manipulated are derived from the result graph reflecting the current state of resources,
2. input data for the transition is derived from the result graph reflecting the current state of resources, and/or
3. the transition is only invoked, if the result graph shows that resources are in a specified state.

✎ **Example 23**

ACME might want to create entries about upcoming concerts in the time line of social network accounts of users that follow ACME. However, ACME marketing suggests that concert information is only posted to users that live in the same town, where a concert takes place.

The links to the time lines of followers of ACME can be found in the resource `sna:acme` representing ACME on the social network (1). Information about the concerts that is used to create the time line entries, are derived from the resources provided by ACME itself (2). New entries (i.e., resources) are created by sending concert information to the time lines by POST, if a user account specifies that a user lives in the town a concert takes place (3).

We use *transition rules* to encode resource manipulations (i.e., transitions in an LDSTS) derived from an RDF graph.

➳ **Definition 15: Transition Rule**

A transition rule $\rho^t : \{B\} \implies \{(m, x, H)\}$ consists of
- a BGP B as rule body
- a tuple as rule head containing
  - a method $m \in \{\mathsf{PUT}, \mathsf{POST}, \mathsf{DELETE}\}$
  - a variable or URI $x \in \mathcal{V} \cup \mathcal{U}$
  - a potentially empty BGP H

If x is a variable $x \in \mathcal{V}$, then it has to appear in the rule body $x \in \mathcal{V}_B$. Every variable in the head H has also to be part of body B, i.e., $\mathcal{V}_H \subset \mathcal{V}_B$.
The execution of a transition rule $\rho^t : \{B\} \implies \{(m, x, H)\}$ over a graph G derives a set of requests $\mathtt{Req}_{\rho^t}$, where for every result binding of BGP B from graph G a request is constructed such that the addressed resource is either the given URI or the binding $\mu(x)$. The payload of the request is the

graph resulting from the substitution of all elements $e \in (E)_H$ in BGP H according to a pattern instance matching $P^{\iota}_{\mu}(e)$ for a random and unique RDF instance mapping ι for every request.

$$\forall \mu \in \Omega_G(B) : \begin{cases} (m, x, P^{\iota}_{\mu}(\mathcal{E}_H)) \in Req_{\rho^t} & \text{if } x \in \mathcal{U} \\ (m, \mu(x), P^{\iota}_{\mu}(\mathcal{E}_H)) \in Req_{\rho^t} & \text{if } x \in \mathcal{V} \end{cases}$$

We denote f^t as the function that maps a transition rule and a graph to the set of derived requests resulting from the execution of the transition rule:

$$f^t(\rho^t, G) = Req_{\rho^t}$$

Intuitively, a transition rule describes how from a given graph requests should be constructed. Result bindings for the BGP in the rule body are used to substitute variables in the rule head to identify the target resource and payload of a request. Consequently, the head of a rule corresponds to an update function of an LDSTS in that it describes an HTTP method that is to be applied to a resource. Thus, a transition rule allows to dynamically infer desired resource changes derived from a given graph.

If no result bindings for the BGP in the rule body are found, no request can be constructed. Therefore, the rule bodies allow to express an intention under which condition an unsafe method is to be applied to a resource.

We want applications to derive their manipulations dynamically from the available information about the current state of resources they interact with. Therefore we extend linked programs to LD-Fu programs:

Definition 16: Linked Data-Fu Program

A Linked Data-Fu program $\mathcal{P}^{Fu} = (G, R, P^d, P^r, P^t)$ is a tuple with

- G a finite set of initial triples, i.e., the starting graph,
- R a finite set of initial resources,
- P^d a finite set of deduction rules,
- P^r a finite set of request rules,
- P^t a finite set of transition rules,

where $G \neq \emptyset \vee R \neq \emptyset$, i.e., there has to be at least one triple in the starting graph or one initial resource specified.

The execution of a Linked Data-Fu program implies in a first step that a result graph $G_{\mathcal{P}}$ is calculated equivalently to a linked program, i.e., the initial resources are retrieved and the returned triples are added to the starting graph G. The rules P^d and P^r are recursively executed over G until the fixpoint is reached.

In a second step after the fixpoint is reached, all transition rules are executed over the result graph $G_{\mathcal{P}}$, which results in a set of derived requests $Req_{\mathcal{P}}$ to manipulate resources:

$$Req_{\mathcal{P}} = \bigcup_{\rho^t \in P^t} f^t(\rho^t, G_{\mathcal{P}})$$

The order in which the request rules are evaluated does not affect $\mathtt{Req}_{\mathcal{P}}$, as manipulating requests do not modify the result graph. Specifically, responses to unsafe methods are considered to contain an empty payload (see Definition 14)[a].

[a]For simplicity we assume POST requests to have empty response payloads. However, the response of POST requests could be included in the initial graph of a subsequent execution of a LD-Fu program (see Section 4.4.3)

LD-Fu programs enable programmers to define their desired state transitions in a declarative rule-based execution language. The transition rules specify the interaction of a client application with REST-based Linked Data resources and congruously a path through the LDSTS. Further, LD-Fu programs allow to specify the conditions under which a specific transition is to be invoked as subject to the states of resources.

Concretely, LD-Fu programs establish the relation between the available information on the current state of resources and the decision on effected changes by leveraging the result graph $G_{\mathcal{P}}$. Specifically, the BGP B in the body of a transition rule is used to dynamically

1. identify the resource to which an HTTP method has to be applied, i.e., leveraging hypermedia controls, as found in $G_{\mathcal{P}}$,
2. derive input data from $G_{\mathcal{P}}$,
3. determine if the conditions under which a transition is to be invoked hold true according to $G_{\mathcal{P}}$.

Regarding 1: Instead of specifying the addressed resource r of a request rule explicitly as URI, a variable x can be used. For every solution binding μ of B from $G_{\mathcal{P}}$, a resource to address is identified by $\mu(x)$. Thus, LD-Fu programs preserve the flexibility implied by REST, as an application does not require the URIs of the resources to change a priori, but can discover them via link traversal. However, we retain the possibility to directly specify a URI.

Regarding 2: Instead of specifying the input payload D^{req} of a request explicitly as RDF graph, a graph pattern H can be used. For every solution binding μ of B from $G_{\mathcal{P}}$, a payload is constructed by substituting every variable in H according to the mapping μ. Thus, LD-Fu programs can dynamically derive input payload to identify different manipulating requests, depending on the available information $G_{\mathcal{P}}$ about the current state of resources. However, per Definition 5 an RDF graph is also a valid BGP, thus the input payload can also be directly specified, where no substitution has to take place in H.

Regarding 3: Instead of defining the manipulations in a fixed manner (i.e., to be executed in any case), the BGP B establishes a condition which has to hold with respect to $G_{\mathcal{P}}$ for requests to be derived. As requests are only derived for every solution binding μ of B from $G_{\mathcal{P}}$, no requests are derived if no solution binding can be found. Not every variable $\mathcal{V}_B$ in the BGP of the rule body has to be used in the rule head, thus arbitrary conditions over $G_{\mathcal{P}}$ can be defined, that are not related to the identification of the resources to address and the construction of the input payload. However, as every variable in the rule head

has to appear in the rule body B at least enough information has to be present to derive the URI of the target resource and the payload for a request.

Apart from the transition rules is the expressive power with regard to data processing of LD-Fu programs similar to Datalog [18].

The approach to establish a local graph that represents the available information on which an application can base its decisions is motivated by the idea that clients have to maintain a knowledge space, in which they store their knowledge about the states of the resources they interact with [61, 99]. Here, the knowledge space is equivalent to the monotonically growing result graph $G_{\mathcal{P}}$, which is filled with the RDF data the client receives after retrieving a resource, as defined by the output function of $\text{LDSTS}_{\mathcal{P}}$.

Additionally many parallels can be identified of LD-Fu programs and the Believe Desire Intention (BDI) software model for intelligent agents as outlined by Rao and Georgeff [92]. Rao describes the BDI model as suited systems acting in environments characterised as follows: The system can have different objectives to accomplish. The environment can at any point in time evolve in many different ways and allows systems to perform many different actions, which achieve the objectives depended on the state of the environment. Further the system might not be capable of fully determining the state of the environment. The rate at which the system can carry out computations and actions is within the bounds of the rate at which the environment changes[75]. The described system–environment architecture strongly resembles the scenario of applications interacting with Web resources, with which we are concerned.

According to Rao, systems adhering to the BDI model have to acquire information about the current state of the environment by performing a series of sensing actions. Such information is called *belief*, and is congruent to the result graph $G_{\mathcal{P}}$ established by a LD-Fu program with a series of lookups (i.e., sensing actions). Further, the system requires information about goals to accomplish, called *desire*. From these goals, together with the information about the current state of he environment, the system can derive the actions to be performed to accomplish the objectives of the system. Such actions are interpreted as choices in a decision tree. LD-Fu programs do not strictly contain goals to achieve, but the transition rules encode in a declarative manner what resource states are desired subject to conditions of the current state of resources. LD-Fu derives requests (i.e., actions) from the acquired current state to achieve the desired resource states.. Thus, the transition rules determine the behaviour of the application similar to the goals in the BDI model. Hence, the described decision tree is akin to the tree of all possible paths through an LDSTS[76], where every request is a decision to transition into a new state.

The abstract architecture of a BDI interpreter, as outlined by Rao, implies that a system should derive a set of options for intended actions from which a subset has to be chosen. We cover the choice which of the derived requests of a LD-Fu

[75] When Interacting with Web resources the rate at with resources change, might exceed the rate at which an application can carry out requests. However, both rates can vary a lot and depend on the Web resources and the hardware on which a client application is executed.

[76] In the decision tree described by Rao the nodes represent decisions, in contrast in an LDSTS the nodes are states and the edges represent the decisions on how to change the state.

program are actually carried out in Section 4.4.2 in the context of the potentially non-deterministic behaviour of LD-Fu programs. Further, the algorithm of an BDI interpreter implies that a system carries out multiple rounds of acquiring information about the current state and deriving intentions. We describe the repeated execution of a LD-Fu program in Section 4.4.3.

Like deduction and request rules, we serialise transition rules with N3 syntax, leveraging W3C vocabularies to describe HTTP messages and methods. In particular we define the BGP H in a rule head of a transition rule as nested triples that represent the object of the property `http:body`. Examples of transition rules are the rules (2), (4), and (5) in Listing 10.

Listing 10: Rules of an LD-Fu program constituting an information dissemination system.

```
# (1) Request rule to retrieve resource representing ACME
{ ?x rdf:type p:InfoItem . } => { [] http:mthd httpm:GET;
                                    http:requestURI acme:acme .} .

# (2) Transition rule to create new blog posts
{ ?x rdf:type p:InfoItem .
  ?x p:content ?c .
  acme:acme p:dissChannel ?mb .
  ?mb rdf:type p:MicroBlogTimeline . }
              => { [] http:mthd httpm:POST;
                      http:requestURI ?mb ;
                      http:body { [] rdf:type mb:Message ;
                                     sioc:content ?c . } } .

# (3) Request rule to retrieve social network account of ACME
{ ?sid rdf:type p:SocialNetworkId . } => { [] http:mthd httpm:GET;
                                             http:requestURI ?sid .} .

# (4) Transition rule to send message to every found follower on SNA
{ sna:acme sna:hasFan ?fan .
  ?x rdf:type p:InfoItem .
  ?x p:content ?c . }
              => { [] http:mthd httpm:POST;
                      http:requestURI ?fan ;
                      http:body { [] rdf:type sna:Message ;
                                     sioc:content ?c . } } .

# (5) Transition rule to send message to every found follower on SNB
{ snb:acme snb:followedBy ?fo .
  ?x rdf:type p:InfoItem .
  ?x p:content ?c . }
              => { [] http:mthd httpm:POST;
                      http:requestURI ?fo ;
                      http:body { [] rdf:type snb:PrivateMsg ;
                                     snb:text ?c . } } .
```

✎ **Example 24**

The IT department of Acme creates the dissemination system with four rules as shown in Listing 10 rules (1)–(4). The marketing department has simply to create new InfoItems and add them to the initial graph G of the program. The system automatically distributes the information over the dissemination channels of Acme. The rules are defined as follows:

1. Whenever an InfoItems is found, retrieve the resource `acme:acme` to get an up-to-date list of the current dissemination channels.
2. If a `p:MicroBlogTimeline` is found (from the retrieved dissemination channels), post a new entry to the time line using the content from the InfoItems.
3. If a social network ID of Acme is found (from the retrieved dissemination channels), retrieve the representation of Acme from the social network to get a list of Acme's followers.
4. Post to every found follower of Acme on SNA a message with the content of the InfoItems.

The described rules disseminate new information items automatically to social network SNA and the micro blog MB.

IT deploys the dissemination system so that marketing can input new InfoItems, i.e., defining the initial graph G, and run the LD-Fu program. The LD-Fu program will retrieve `acme:acme` to get all dissemination channels with rule (1). For every found social network account, information of ACME are retrieved with rule (3) to acquire a list of followers on the social network. After all resources are retrieved the fixpoint of the program is reached and a result graph with all relevant information is established.

With the transition rules (2) and (4) new posts on the micro blog and entries in the time lines of followers are created, derived from the result graph of the program.

Other dissemination channels can easily be added to the system, simply by adding corresponding rules in the system. For example, we consider that IT adds support for social network SNB by adding transition rule (5) that uses SNB's vocabulary for retrieving followers and sending a message[a].

The new dissemination channel is active when marketing overwrites (PUT) `acme:acme` to include ACME's identifier in SNB as dissemination channel, typed as `p:SocialNetworkId`.

[a] The vocabulary and structure of messages accepted as request payload can be derived from descriptions as covered in Chapter 5

4.4.1 *Program Execution*

For the execution of a LD-Fu program we can extend the system described in Section 3.3. The BGPs of transition rule bodies are included in the operator plan in a straight forward manner. In particular the TriplePattern and EquiJoin operators become subject to the overall optimisations to remove duplicate operators.

To include the head of transition rules we extend the Request operator to also be used for transition rules. Request operators can still construct lookups, which are immediately and in parallel executed once a result binding is found. Additionally, in the case of a transition rule, Request operators construct requests with the specified HTTP method and the payload by substituting the variable in the BGP of the rule head.

The lookup component is extended to be able to execute all kinds of requests (i.e., it becomes a request component). The identified requests resulting from transition rules are passed to the request component, where the requests are stored until the program has reached its fixpoint. After the result graph is determined and all manipulating requests are derived, these requests are executed in parallel by the RequestWorker threads. Under some circumstances the manipulating requests can also be executed in parallel with the CPU-bound tasks of the TripleWorker threads and the lookups, i.e., before the fixpoint is reached (see Section 4.4.2).

The request component is able to send messages as input payload of manipulating requests and receive data from lookups in various RDF serialisations. The RDF graphs in different serialisations are mapped to and derived from the internally used Java object model for triples and bindings of our system. Thus, the system is capable of interacting with Linked APIs independently from the employed RDF serialisation of the API. The transformation of serialisations is similar to the functionality that could be provided by a wrapper for the API (see Section 4.2.4). However, the transformation does not include an additional mapping to a vocabulary or ontology, as the vocabulary used in the original RDF serialisation of the API is preserved. Thus, the transformation can be done completely automated without any additional manual intervention.

An LD-Fu program terminates when the result graph is constructed (i.e., a fixpoint is reached) and the derived manipulations are executed (or a subset of the derived manipulations, cf. Section 4.4.2). The determination of the fixpoint in an LD-Fu program is equivalent to the determination of a fixpoint in a linked program, i.e., request and deduction rules are applied recursively until no new triples can be derived and no further requests are identified to lookup. Thus, given that the set of retrieved resources $R_{\mathcal{P}}$ is finite, an execution of a LD-Fu program is guaranteed to reach a fixpoint with a finite result graph (see Section 3.2). As the result graph is finite, the set of requests derived via transition rules from the result graph is also finite. The transition rules are not executed recursively, i.e., the payload of responses to manipulating requests are not processed by the operator plan. In particular, an LDSTS assumes that such responses have an empty payload. Consequently, the termination of a single execution of a LD-Fu program is also guaranteed, if $R_{\mathcal{P}}$ is finite.

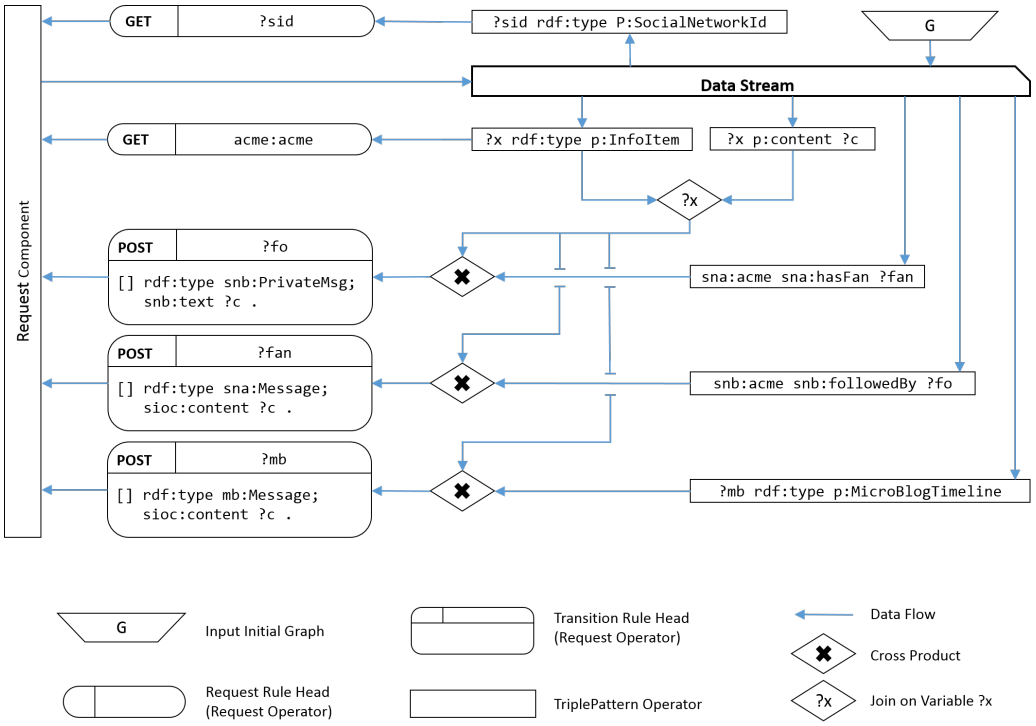


Figure 22: Dataflow of ACME's dissemination system

The overall architecture establishes a data flow similar to a Rete algorithm, where the TriplePattern operators form the alpha network and the rest of the operator plan establishes the beta network. The triples and tuples of binding results are working memory elements in the data flow system and are processed in a streaming fashion.

✎ Example 25

Figure 22 illustrates the data flow of ACME's dissemination system as it is established by the rules in Listing 10. Due to the optimisations of the operator plan the join on `?x` is re-used, i.e., has multiple outgoing edges. The data stream receives data from the initial graph G and the request component after executing lookups. The data stream is in the operator plan buffered by the TripleQueue. The initial graph contains InfoItems, which trigger the retrieval of `acme:acme` containing links to the dissemination channels. The social networks fire a rule, which retrieves the social network id's of ACME and thus retrieve the corresponding followers. Both, social network followers and micro blog time lines trigger the corresponding POST transaction rules that sent the information in the appropriate vocabulary to the dissemination channels, i.e., as micro blog posts or personal messages to the followers.

4.4.2 *Non-Deterministic Behaviour*

The final effect of an execution of a LD-Fu program on the remote resources is not necessarily deterministically defined, i.e., the final state of the $\text{LDSTS}_{\mathcal{P}}$ after executing a LD-Fu program cannot a priori be predicted even if we assume that resource do not change on their own or are changed by other applications. Such a non-deterministic behaviour is described in Example 26.

✎ **Example 26**

Marketing proposes to implement more intricate strategies to maintain already created entries on the time lines of followers of ACME on social networks. In particular, a user can opt-out to receive information from ACME on social networks. If a user opts-out a resource is created representing the users opt-out statement and all entries of ACME on the user's time line should be deleted. Further, if a user buys a ticket for a concert, the entry on the user's time line informing about this concert should be extended to include a "thank you" message. Consequently IT extends the dissemination channel with the following transition rules:

1. If a resource is found containing a user's opt-out message, all entries are to be removed via a DELETE request.
2. If a resource is found containing the order of tickets for a concert by a user, the corresponding entry on the user's time line is overwritten via a PUT request.

However, if a user buys a ticket for a concert and also uses the occasion to opt-out of receiving entries on his time line, both rules derive two conflicting requests for the entry about the concert. On the one hand, the entry should be deleted. On the other hand it also should be overwritten. Especially because a PUT also creates a resource, if it does not exist (or was deleted), it remains undetermined if the user has an extended entry or no entry after the next execution of the dissemination system.

To analyse the non-determinism of a given LD-Fu program we adopt the notion of *conflict sets* from the domain of production rule systems [91]. Every conflict set of an LD-Fu program contains the derived manipulating requests from the result graph $G_{\mathcal{P}}$ that represent conflicting changes on a resource. To describe sets of requests we first must define the equivalence of requests, as a set cannot contain two equivalent requests:

➳ **Definition 17: Request Conflict Set**

Two requests (m_1, r_1, D_1^{req}) and (m_2, r_2, D_2^{req}) are equivalent, iff

- the methods are equivalent, $m_1 = m_2$,
- both target resources refer to the same information resource, $r_1 \xrightarrow{i} T^u$, and $r_2 \xrightarrow{i} T^u$
- an RDF instance mapping exists, such that both payloads contain the same triples, $\forall \langle s, p, o \rangle \in D_1^{req} \exists \iota : \langle \iota(s), \iota(o), \iota(o) \rangle \in D_2^{req}$

Let $Req_{\mathcal{P}}$ be the set of all requests derived from the result graph $G_{\mathcal{P}}$ of LD-Fu program $\mathcal{P}^{Fu}$.
A request conflict set $CS^{u}_{\mathcal{P}}$ of LD-Fu program $\mathcal{P}^{Fu}$ for a resource r is a subset of all derived manipulating requests ($m \in \{PUT, POST, DELETE\}$) that modify the same information resource u, i.e.,

$$CS^{u}_{\mathcal{P}} = \{(m_i, r_i, D^{req}_i) | \forall r_i : r_i \xrightarrow{i} T^{u}\} \qquad CS^{u}_{\mathcal{P}} \subseteq Req_{\mathcal{P}}$$

We denote $\Gamma_{\mathcal{P}}$ as the set of all conflict sets of a LD-Fu program $\mathcal{P}^{Fu}$.

$$\forall (m, r, D^{req}) \in Req_{\mathcal{P}} : \; CS^{u}_{\mathcal{P}} \in \Gamma_{\mathcal{P}}, \text{ if } r \xrightarrow{i} T^{u}$$

Intuitively, a request conflict set contains all manipulating requests that target the same resource. If a conflict set contains more than one request, the corresponding LD-Fu program has derived different actions to change the same resource and the ultimate effect of the program is not deterministically defined. If all request conflict sets of a LD-Fu program contain exactly one element $\forall CS^{u}_{\mathcal{P}} \in \Gamma_{\mathcal{P}} : \; card(CS^{u}_{\mathcal{P}}) = 1$, the outcome is deterministic, as every resource that is manipulated is addressed with a unique request.

It is noteworthy, that two requests with the method POST in a conflict set (i.e., two POST requests with different input payload at the same resource), do not necessarily represent a conflict. A request with the POST method can have no effect or an additive effect on the addressed resource, e.g., adds the triples to the addressed resource. Thus, multiple POST requests can affect a resource independently from the order of execution, as POST is not defined to be idempotent. However, the specification for POST allows to change resources in an arbitrary way (see Section 4.3) and therefore might result in conflicting changes. As a client application cannot a priori predict if multiple POST requests on the same resource represent a conflict, we include POST requests in conflict sets in a non-discriminatory manner.

Further, only successful requests on the same resource can constitute conflicts, as unsuccessful requests (with response codes 4xx and 5xx) do not change resources. However, as client application cannot a priori predict, if a request will be successful, we have to assume all derived requests will be successful and potentially cause a conflicting change.

Several strategies can be employed to handle conflicting requests. Such strategies can be employed to resolve or mitigate the non-deterministic behaviour of a LD-Fu program. In the following we describe approaches that can be used in the face of conflicting requests:

NAIVE All derived requests $Req_{\mathcal{P}}$ are executed independently of conflicts. Such a naive approach fully accepts the potentially non-deterministic behaviour.

A naive approach can decrease the runtime of an application, as the results graph $G_{\mathcal{P}}$, does not have to be completely constructed, before a manipulating request is executed. Rather, as soon as a manipulating request is derived it can be executed, similar to the lookups derived by request rules,

as we do not require to establish the conflict sets.[77] Thus, the derived requests can be executed in parallel with the CPU-bound tasks of the data processing.

However, the naive approach might lead to unnecessary requests in the sense that requests are executed that result in changes that are undone by subsequent requests, e.g., if the same resource is repeatedly overwritten. Unnecessary requests can negatively influence the overall runtime of the program. The final state of a resource addressed by multiple requests, depends on the order in which the requests are executed.

RANDOM From every request conflict set $CS^u_{\mathcal{P}} \in \Gamma_{\mathcal{P}}$ exactly one, randomly chosen request is executed. A random choice still accepts the potentially nondeterministic behaviour of program $\mathcal{P}^{Fu}$.

Similar to the naive approach requests can be executed without waiting for the complete construction of the result graph $G_{\mathcal{P}}$: As soon as a request is derived it is executed, where a list of all addressed resources is maintained. If a second manipulating request on the same resource is derived it will not be executed.

For all conflict sets $CS^u_{\mathcal{P}}$ that only contain requests with idempotent methods $\forall(m, r, D^{req}) \in CS^u_{\mathcal{P}} : \quad m \in \{PUT, DELETE\}$, the final state of the addressed resource is equivalent to the naive approach: With the naive approach the final state only depends on the last executed idempotent request on a resource, which could also be chosen in the random approach. However, due to the potentially arbitrary changes of POST requests, the effect of the random approach for conflict sets containing POST requests might differ from the naive approach.

The random approach guarantees that every manipulated resource is only changed once. Thus no unnecessary requests are executed.

NONE Every resource u, for which a conflict set exists with more than one element $card(CS^u_{\mathcal{P}}) > 1$, is not changed, i.e., no manipulating request is executed if conflicting requests are derived.

To not affect resources for which derived requests are in conflict ensures a fully deterministic behaviour. No unnecessary requests are executed, as every changed resource is only manipulated once.

To apply the approach all conflict sets have to be completely established, i.e., manipulating requests can only be executed after a LD-Fu program has reached the fixpoint and the result graph $G_{\mathcal{P}}$ is established, from which all manipulating requests $Req_{\mathcal{P}}$ are derived. Only after all request conflict sets are completely established, the client application can determine with certainty the cardinality of each conflict set, and execute only the requests from conflict sets with with exactly one element.

[77] If a resource is manipulated, it must not be retrieved via a request rule anymore, as this could cause the program to make decisions on already changed resources and thus change the execution semantics.

STRATIFICATION In every request conflict set $CS^{u}_{\mathcal{P}}$ a distinct request is chosen according to some weak ordering $\overset{o}{>}$. An ordering $\overset{o}{>}$ establishes a preference relation between pairs of requests, such that for any requests $j, k, l \in CS^{u}_{\mathcal{P}}$ the following holds true:

- $\forall j: \; j \overset{o}{\not>} j$ (irreflexivity)
- $\forall j, k, l: \;$ if $j \overset{o}{>} k$ and $k \overset{o}{>} l$ then $j \overset{o}{>} l$ (transitivity)
- $\forall j, k, l: \;$ if $(j \overset{o}{\not>} k) \wedge (k \overset{o}{\not>} j)$ and $(k \overset{o}{\not>} l) \wedge (l \overset{o}{\not>} k)$ then $(j \overset{o}{\not>} l) \wedge (l \overset{o}{\not>} j)$ (transitivity of incomparability)

From the requests in a conflict set we choose a request j to execute, such that no other request is preferred over j, i.e., $\nexists k \in CS^{u}_{\mathcal{P}}: \; k \overset{o}{>} j, k \neq j$. However, we can only grantee that a chosen request is unique if $\overset{o}{>}$ establishes a total ordering for all pairwise disjoint requests in the request conflict set, i.e., no two requests are incomparable.

$$\exists! j \nexists k \in CS^{u}_{\mathcal{P}}: \; k \overset{o}{>} j, k \neq j \iff \forall k, l \in CS^{u}_{\mathcal{P}} \text{ with } k \neq l: \; (k \overset{o}{>} l) \vee (l \overset{o}{>} k)$$

Thus, there is a distinct request in a conflict set, which is preferred over all other requests. Multiple possible orderings are possible, examples include:

- Order via request method: An ordering of the request methods can be defined, from which the preference of the requests in conflict sets can be derived. E.g., DELETE could be preferred over PUT, which is preferred over POST. If a conflict set contains a DELETE request, this request would be preferred over all other requests. Similarly, if a conflict set does not contain a DELETE request, but a PUT request, the PUT request would be preferred. However, a request conflict set might contain multiple requests with the same method, i.e., multiple requests that are not comparable according to this ordering. Thus, an ordering via request method does not necessarily guarantee a deterministic behavior, as a unique request cannot be chosen in all cases.
- Order via rule priority: The transition rules of an LD-Fu program can have a priority assigned, which determines the order of requests in a conflict set, i.e, requests derived from transition rules with a higher priority are preferred over requests from transition rules with lower priority. However, a request conflict set might contain multiple requests derived from the same transition rule, i.e., multiple requests that are not comparable according to this ordering. Thus, an ordering via rule priority does not necessarily guarantee a deterministic behavior, as a unique request cannot be chosen in all cases.

Multiple stratification strategies can also be combined. However, which strategy is appropriate and results in a deterministic behaviour depends on the application context. In Example 26 the introduced rules could be prioritized in such a way that the removal of time line entries is prioritised over the extension of entries, which might capture the intentions of the marketing department best.

In the case a LD-Fu program is inherently determined, i.e., all conflict sets contain exactly one request, all described approaches are equivalent in terms of the final effect on changed resources. The approaches described above can have a different impact on overall runtime and memory consumption of a LD-Fu program, depending on the number of unnecessarily executed requests (in the case of *naive*), the number of requests that have to be executed after the fixpoint is reached (in the case of none and *stratification*), and the necessity to maintain lists of already performed requests (in the case of *random*). This impact depends on size and amount of conflict sets for a given LD-Fu program. E.g., if a LD-Fu program has only one very large conflict set, the *naive* approach implies that many unnecessary requests are executed causing a large overhead. In contrast, the impact of the fact that *none* and *stratification* have first to completely establish the conflict set is minor, because no or only one request have to be executed after the fixpoint is reached.

To implement approaches *none* or *stratification* a client application would require information to what information resource u a target resource r of a request refers $r \xrightarrow{i} T^u$. However, such information, which is necessary to correctly establish the conflict sets, is not necessarily a priori given. Therefore, a client might have to approximate the conflict sets, by assuming all target resources are information resources, i.e., the request conflict sets contain all derived requests with the same target resource. Such an approximation can cause distinct conflict sets, which contain requests that are referred to the same information resource, i.e., the distinct conflict sets actually should form a joined conflict set. E.g., two conflict sets $CS^{r_1}_{\mathcal{P}}$ and $CS^{r_2}_{\mathcal{P}}$ could be established for the target resources r_1 and r_2, where r_1 and r_2 refer to the same information resource u, i.e., $r_1 \xrightarrow{i} T^u$ and $r_2 \xrightarrow{i} T^u$. The requests in both conflict sets actually belong to a single conflict set $CS^u_{\mathcal{P}}$, as all requests modify the same resource, i.e., all requests represent the same conflict.

The approaches *naive* and *random* do not require to establish the conflict sets to execute a request. With the approach *none*, some resources might be modified even though there are conflicting requests, because of the approximation of conflict sets. Specifically, if a conflict set with a single resource is established, the request is executed, even if requests in other conflict sets refer to the same information resource. Similarly with a *stratification* approach a request might be chosen for execution, even though another request should be preferred, which is not considered in the ordering, because it is in another conflict set.

Finally it should be noted that a deterministic LD-Fu program (either inherently or by use of an appropriate conflict resolution approach) has only a defined outcome for a given result graph $G_{\mathcal{P}}$. In turn, the result graph $G_{\mathcal{P}}$ of a program is deterministically defined for a finite and stable set of retrieved resources. However, we have to assume that resources on the Web are not stable and can change dynamically. Thus, the characteristic of a LD-Fu program to exhibit deterministic behaviour does not necessarily imply that its outcome can be a priori predicted from the program alone. The determinism rather allows to make predictions under what circumstances (i.e., from what state in an LDSTS) what actions will be derived (i.e., what requests will be executed), i.e., a LD-Fu program is capable of handling the entropy of a Linked API resulting from the dynamically changing

resources by reaction on it, but does not reduce the inherent entropy by making the reaction of the Linked API predictable.

4.4.3 *Repeated Program Execution*

The objective of an application can not always be achieved with a single execution of an LD-Fu program. The goal of a program might imply a constant monitoring of the state of some resources to ensure an appropriate reaction as soon as the resource states change, as discussed in Section 3.2, Example 12. Such a monitoring requires the repeated execution of a LD-Fu program, where the frequency of the repetition reflects how closely the resources are monitored.

In other scenarios the goal of a LD-Fu program might require that a resource is modified multiple times, or that a resource is retrieved after it is modified. Similarly, to accomplish the objective in such scenarios an LD-Fu program has to be executed multiple times until the goal is achieved.

✎ **Example 27**

The social network SNB changes its policy regarding the creation of entries in users time lines. Links to other resources cannot anymore be directly included when an entry is created. Rather, an entry with a message has to be first created and can afterwards be overwritten to include links. Such overwrites do not immediately change an entry resource, but are first checked by SNB, if the included links adhere to the terms and conditions of SNB.

To adapt to the new situation ACME's IT devises a new LD-Fu program for the dissemination system. For every new InfoItem created by the marketing department a new entry is created in the time lines of ACME's fans containing only the message of the InfoItem. Once an entry is created, it is overwritten to include the designed links of the InfoItem. After the overwriting request is sent, the entry is monitored, to ensure that the links eventually appear in the entry.

To analyse the temporal behaviour of the repeated execution of LD-Fu programs we require a notion of time. For an application built upon an LD-Fu program $\mathcal{P}^{Fu}$ we define a point in time as a discrete state σ of $LDSTS_{\mathcal{P}}$.

At first, we assume that all resources are stable, i.e., the resources do not change on their own and no other application influences the resources. If a program $\mathcal{P}^{Fu}$ is executed, it derives a result graph from the current state σ_1 and executes manipulating requests, thus advancing time to a new state σ_2, see Figure 23. The execution of the program might imply a series of requests, i.e. $LDSTS_{\mathcal{P}}$ can advance through several states before eventually reaching σ_2. However, as an LD-Fu program $\mathcal{P}^{Fu}$ defines declaratively all changes to be made under given circumstances in a cohesive manner, we consider a single execution of $\mathcal{P}^{Fu}$ as a single step in time from σ_n to σ_{n+1}. We denote $\mathcal{P}(G) \rightarrow \sigma$ as a step in time to state $sigma$, derived from graph $G\mathcal{P}$ by program $\mathcal{P}^{Fu}$.

If a LD-Fu is executed repeatedly, it advances iteratively the current point in time, i.e., the current state of $LDSTS_{\mathcal{P}}$. Thus, at every execution the program takes into account the resources, which were changed by the previous execution.

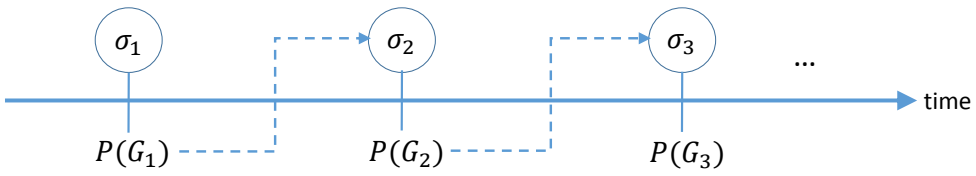


Figure 23: Points of time as distinct states, driven by an LD-Fu program.

However, a repeated execution is stateless, in the sense that no information about previous actions are available, i.e., every execution $\mathcal{P}(G)$ does not depend directly on previous executions, but only on graph $G_{\mathcal{P}}$. We denote $\mathcal{P}^n(\sigma_k) \rightarrow \sigma_l$ as the n-th iterative execution of program $\mathcal{P}^{Fu}$ starting at state σ_k, which results in state σ_l.

The repeated execution of a program $\mathcal{P}^{Fu}$ starting in a state σ_s can result in the following developments of the LDSTS$_{\mathcal{P}}$, given that the behaviour of $\mathcal{P}^{Fu}$ is deterministic (either inherently or due to the application of an appropriate conflict resolution approach):

FIXPOINT The LDSTS$_{\mathcal{P}}$ reaches a fixoint, i.e., a state σ_f is reached after n iterative executions, where every further application of $\mathcal{P}^{Fu}$ does not change the state anymore.

$$\exists n \in \mathbb{N} : \mathcal{P}^n(\sigma_s) \rightarrow \sigma_f \wedge \mathcal{P}^{(n+1)}(\sigma_s) \rightarrow \sigma_f$$

OSCILLATION After a series of iterative executions a state σ_o is reached, over which $\mathcal{P}^{Fu}$ was already executed in a previous iteration. Thus a cycle is triggered, which always leads back to σ_o.

$$\exists n \exists m \in \mathbb{N} : \mathcal{P}^n(\sigma_s) \rightarrow \sigma_o \wedge \mathcal{P}^m(\sigma_o) \rightarrow \sigma_o \text{ with } n < m, m > 0$$

INFINITE EVOLUTION At every point of an iterative execution the continued execution adds new triples (by extending existing resources or creating new resources) to the subsequent states, which are never removed. Thus, every iteration creates a new state.

$$\forall n \exists k \in \mathbb{N} : \mathcal{P}^n(\sigma_s) \rightarrow \sigma_n \wedge \mathcal{P}^k(\sigma_n) \rightarrow \sigma_k \text{ with } k > 0,\ card(\sigma_n) < card(\sigma_k)$$

The repeated execution of an LD-Fu program does not necessarily reach a fixpoint, even if the number of all retrieved resources $Req_{\mathcal{P}}$ at every iteration step is finite. If $Req_{\mathcal{P}}$ is finite, only the construction of the result graph is guaranteed to reach a fixpoint at the current state σ_n, i.e., the next $\sigma_n + 1$ will be reached. The reason for the repeated execution of an LD-Fu program potentially not reaching a fixpoint is that the manipulating requests at any iteration $\mathcal{P}^n(\sigma_n)$ can result in triples with new URIs or literals, which become part of the next state $\sigma_{(n+1)}$. Here, new URIs or literals are terms that are not part of the result graph $G_{\mathcal{P}}$[78]

[78]RDF terms not in the result graph, have to be terms that are not in the initial graph G, the request and deduction rules or any of the retrieved resources $Req_{\mathcal{P}}$, see Section 3.2.

Specifically POST requests can result in the creation of new resources, identified with a URI chosen by the server, i.e., a new URI not part of the result graph. Further, POST requests can change resources to include URIs and literals that are not part of the request payload. Thus, at every iteration of the repeated execution a new result graph might be derived from the new state σ, which again can again lead to the creation of triples with new URIs or literals.

Resources can be deleted and created. If a repeated execution of a program causes at any point all previously deleted resources to be created again without adding any new triples the states, the program will permanently cycle through the same set of states, thus have an oscillating development. E.g., consider a program $\mathcal{P}^{Fu}$ that derives exactly two manipulating requests: a request to DELETE resource r, if r exists and a request to create the same resource r via PUT. Further, $\mathcal{P}^{Fu}$ operates with a stratification conflict resolution, where DELETE requests are always preferred over PUT requests. At every iteration of the repeated execution of $\mathcal{P}^{Fu}$ either r is created or deleted, thus the program cycles between two states.

If at every iteration of a repeated execution a state is reached, which was not reached at any previous iteration, the repeated execution causes a continuous evolution of the $LDSTS_{\mathcal{P}}$. An infinite evolution development is in particular possible due to POST requests, which can change addressed resources in an arbitrary manner. E.g., consider a program $\mathcal{P}^{Fu}$ that retrieves a resource r, which represents a counter as a natural number. If the counter is found in the result graph $G_{\mathcal{P}}$, the program executes a POST request at r, which results the counter to increase by 1. Thus, every iteration of a repeated execution of $\mathcal{P}^{Fu}$ increases the counter[79].

✎ **Example 28**

The new LD-Fu program for the dissemination system of ACME (see Example 27) contains the transition rules shown in Listing 11. To create entries on time lines of social network SNB the program has to be executed repeatedly.

As the initial creation of an entry on SNB happens via POST request, IT has to prevent that a new entry is created at every iteration of the repeated execution, i.e., an infinite evolution of the state has to be prevented. As a repeated execution is stateless, the InfoItems are extended: InfoItems now contain an additional triple, which details the state of the InfoItems with respect to the entries in SNB. Thus, the InfoItems which are created by the marketing department now contain the message content, a relevant link, and the status with respect to SNB:

```
acme:infoItem37 p:content "Upcoming concert..." .
acme:infoItem37 rdfs:seeAlso acme:concert84 .
acme:infoItem37 p:snbState "not created" .
```

Furthermore, InfoItems are retrievable and changable URI-identified resources. Specifically, InfoItems can be adressed with POST requests to specifically change the SNB status of the InfoItems. Thus, at ever iteration the

[79] The arithmetic calculation of the successor of the current counter position is done server-side.

dissemination system has information available, if a corresponding entry has to be created or is already present on time lines of SNB.

At every iteration all InfoItems (as specified by marketing in the initial requests), all time lines of followers of ACME on SNB, and all entries on the time lines are retrieved to build a result graph. To create entries on SNB the LD-Fu program of the dissemination system contains five transition rules:

- Rule (1): If an InfoItems is found, which shows that a corresponding entry has not been created, an entry is created with a POST request.
- Rule (2): The state of an InfoItems is changed from "not created" to "created". Rule (2) is always triggered together with rule (1) in the same iteration, as the BGP in the body of rule (2) is a subset of the BGP of rule (1). Thus, the state of the InfoItems is changed, when an entry is created.
- Rule (3): If an entry `?ent` is found, with the same message content `?c` as an InfoItems, which shows to be "created" on SNB, then `?ent` is overwritten with the same message content and the links designated by the InfoItems.
- Rule (4): The state of an InfoItems is changed from "created" to "link submitted". Rule (4) is always triggered together with rule (3) in the same iteration, as the BGP in the body of rule (4) is a subset of the BGP of rule (3). Thus, the state of the InfoItems is changed, when an entry is extended to contain a link.
- Rule (5): Finally, if an InfoItems shows that its link was submitted and an entry actually contains the link of the InfoItems, the state of the InfoItems is changed to "done". As the dissemination system keeps repeatedly executing the program, rule (5) implements the monitoring of the entry resource to report if the link is eventually added[a].

Every transition rule depends in the rule body on the state of the InfoItems. The repeated execution continuously advances the state of the InfoItems from "not created" over "created" to"link submitted" and potentially "done", if the extension with a link is successful. Assuming that no additional InfoItems are created (or added to the initial requests), the repeated execution of the program converges at a fixpoint, where for every InfoItems an entry is created. At the fixpoint every InfoItems is either in state "done" (if a link was added) or in state "created" (if no link can be added).

[a] Although the change of the entry to include a link is effectively caused by the PUT request from the dissemination system, from the viewpoint of the LD-Fu program the entry changes on its own, i.e., is not stable, because of the statelessness of the repeated execution.

Listing 11: Transition Rules of an LD-Fu program for the repeated execution to create entries on a social network

```
1
2 # (1) Create entry to every found follower on SNB
3 { snb:acme snb:followedBy ?fo .
4   ?x p:snbState "not created" .
```

```
  ?x p:content ?c . }
               => { [] http:mthd httpm:POST;
                       http:requestURI ?fo ;
                       http:body { [] rdf:type snb:PrivateMsg ;
                                      snb:text ?c . } } .

# (2) Change status resource of InfoItem
{   ?x p:snbState "not created" . }
               => { [] http:mthd httpm:POST;
                       http:requestURI ?x ;
                       http:body { ?x p:snbState "created" . } } .

# (3) Create entry to contain link
{ ?fo snb:containsEntry ?ent .
  ?ent snb:text ?c .
  ?x p:snbState "created" .
  ?x p:content ?c .
  ?x rdfs:seeAlso ?link . }
               => { [] http:mthd httpm:PUT;
                       http:requestURI ?ent ;
                       http:body { [] rdf:type snb:PrivateMsg ;
                                      snb:text ?c ;
                                      snb:link ?link . } } .

# (4) Create entry to contain link
{  ?x p:snbState "created" . }
               => { [] http:mthd httpm:POST;
                       http:requestURI ?x ;
                       http:body { ?x p:snbState "link submitted" . } } .

# (5) Create entry to contain link
{  ?x p:snbState "link submitted" .
   ?x rdfs:seeAlso ?link .
   ?ent snb:link ?link .}
               => { [] http:mthd httpm:POST;
                       http:requestURI ?x ;
                       http:body { ?x p:snbState "done" . } } .
```

4.5 EXPERIMENTS

In the following we describe the results of two experiments to evaluate the performance of our system for the execution of LD-Fu programs to realise applications with low runtimes. Specifically, we are interested in the behaviour of the system in the face of manipulations. In particular, we evaluate the overall runtime of a single program execution given an increasing amount of resources that are manipulated by the program, as well as an increasing amount of transition rules (see Section 4.5.1). We compare the runtime of a single program runs with the runtime of Cwm, a Semantic Web data processor. Further, we evaluate the runtime behaviour of our system under repeated program executions by implementing Conway's Game of Life [36] (see Section 4.5.2).

All experiments are conducted on the following local machine:

SETUP 1 Lenovo Thinkpad with an Intel i7–4600U processor with 2 physical cores at 2.10 GHz and 12 GB main memory. Hyperthreading is enabled resulting in 4 logical cores. The operating system is Arch Linux SMP PREEMPT (2015-05-13) 64bit GNU/Linux with 4.0.3-1-Arch kernel.

Thus, we evaluate our system to execute LD-Fu programs on commodity hardware with the intent to show the performance achieved by the parallel execution architecture not simply on high-end industrial servers.

4.5.1 *Number Retrieval and Manipulation*

We deploy a sets of consecutive numbers as Linked Data resources. Every resource represents a natural number. We deploy the resources used locally to minimise execution time variations caused by network latency. Retrieving a number resource returns three triples, as shown in Listing 12. Resources can also be overwritten with a PUT request. We use namespace prefix `local` for the deployed numbers. Every number resource is typed as `number`, contains its value as literal, and a link to the successor of the number:

Listing 12: RDF graph of a resource representing a natural number.

```
1 local:1 rdf:type gol:Number .
2 local:1 local:value "1" .
3 local:1 local:successor local:2 .
```

We choose this design to easily keep track of the amount of performed interactions.

For the evaluation we start with the resource number 0 in the initial requests. We identify and retrieve the successor of the number. The successor of a number yields a new successor to retrieve. Thus, we retrieve continuously the complete set of number resources. Once we retrieve a number resource we overwrite the resource with a triple, which indicates that the resource was retrieved. The interactions of this set-up are illustrated in Figure 24.

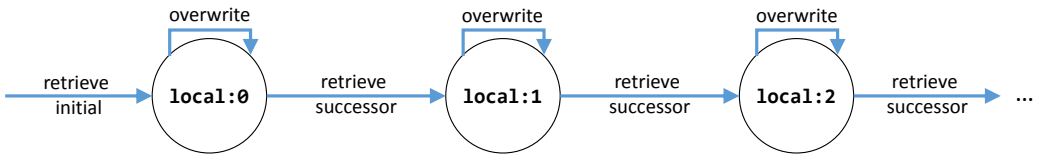


Figure 24: Interactions with a set number resources.

We implement the described scenario with an LD-Fu program and measure the necessary runtime to overwrite all number resources in sets of different sizes. We compare the runtime for number resource sets of different size with an implementation in Cwm[80] version 1.2.1, a data-processor for the Semantic Web. Cwm uses a local triple store that supports the full N3 language to save data and intermediate results. The local triple store of Cwm uses seven indices to allow for

[80] http://www.w3.org/2000/10/swap/doc/cwm.html; retrieved 2015-04-10.

a rapid readout of the local data with almost every combination of subject, predicate and object patterns. If data is added on-the-fly, the index structures have to be updated. For inferencing Cwm uses a forward chain reasoner for N3 rules. The pattern matching for the rules is done by recursive search with optimisations, such as identifying an optimal ordering for the evaluation of the rules and patterns.

Cwm is built as a general purpose tool to query, process, filter and manipulate data from the Semantic Web, i.e., it is focused on the interaction with Linked Data. As such, the motivation behind Cwm is close to LD-Fu programs. However, Cwm is not targeted on the direct manipulation of Web resources according to REST principles, but only their retrieval and the local manipulation of the data. Therefore, we cannot overwrite the number resources with Cwm, but only retrieve the data and identify the numbers. An application, in which Cwm is integrated, would have to use the information about the number resources and implement the manipulating request separately. For our evaluation, we exclude the time necessary for manipulations when using Cwm, i.e., with Cwm we measure only the time to perform lookups and evaluate queries to realise the necessary link traversal, while with LD-Fu we also perform the manipulations and include the required time in the measurements.

We realise the interactions with a LD-Fu program and two Cwm implementations:

LD-FU For LD-Fu we use a request rule and a transition rule as shown in Listing 13.

We add the first number `local:0` to the initial requests. If a number `?n` with value and successor is found, request rule (1) retrieves the successor `?succ` of the number. Thus, all numbers are found and retrieved in an iterative manner in one program execution. Rule (2) overwrites any found number `?n` with an new triple, if it has a value. For the execution we employ the *blocking multi* threading model with four TripleWorker and eight RequestWorker threads (see Section 3.4). The program is inherently deterministic, as every resource is only manipulated once with rule (2). Therefore, we use the *naive* conflict resolution approach, which allows us to execute manipulating requests as soon as they are found, rather than first constructing the complete result graph with all numbers.

Listing 13: Request and transition rule to retrieve and overwrite number resources.

```
#(1) Retrieve successor
{ ?n rdf:type local:Number .
  ?n local:value ?val .
  ?n local:successor ?succ . } => { [] http:mthd httpm:GET;
                                      http:requestURI ?succ . }.

#(2) Overwrite Number
{ ?n rdf:type local:Number .
  ?n local:value ?val . }
                => { [] http:mthd httpm:PUT;
                        http:requestURI ?n ;
                        http:body { ?n local:status "seen" . } .  }.
```

CWM DIRECT Cwm offers built-in functions to perform Web-aware queries in rules. The keyword `log:semantics` in a BGP of a rule allows to resolve a URI and bind the retrieved RDF graph to a variable as formula. The formula bound to a variable can then be used to construct triples in the rule head. Listing 14 shows the Cwm rule to retrieve all numbers.

Like in the approach for the LD-Fu program we us a BGP to identify a number `?n` with value and successor, however we have to expect this graph to appear nested in the subject of a triple (lines 2–4). The successor is marked to be retrieved (line 5) and bound as formula in subject position to a new triple (line 6) that is written to the triple store. Cwm recursively applies the rule to the triple store, thus retrieving all numbers.

Since a retrieved number representation can only be bound as formula in triples (i.e., nested in subject or object position of a triple), we have to employ a rule with nested BGPs in the rule head, making the evaluation of the rule more complex than in the case of the LD-Fu program.

Listing 14: Rule in N3 Cwm syntax to retrieve all numbers via link traversal.

```
1 #(1) Retrieve successor
2 { { ?n rdf:type local:Number .
3     ?n local:value ?val .
4     ?n local:successor ?succ .}  local:status  "seen".
5     ?succ log semantics ?sem . }
6                       => { ?sem local:status  "seen". }.
```

CWM IMPORT To compare the performance of Cwm with the LD-Fu, with rules bodies that are equally complex, we implement the retrieval of the number resources with another Cwm rule shown in Listing 15.

We use the same BGP in the Cwm rule body to identify the successor of a number as in the request rule of the LD-Fu program. For every found match Cwm writes a triple in its store, with predicate `owl:imports` and the successor as object. Cwm offers a command to retrieve all resources in object position of triples with `owl:imports` as predicate. Thus, we programmatically instruct Cwm to apply the rule and retrieve the successors, as many times as needed. However, the *Cwm import* implementation does not have the same functionality as the LD-Fu program: A priori we have to programmatically define how often the rule followed by the retrieve command have to be executed (once for every number), rather than retrieving all identified successors automatically via link traversal.

Listing 15: Rule in N3 Cwm syntax to import the successor of a numbers

```
1 #(1) Retrieve successor
2 { ?n rdf:type local:Number .
3   ?n local:value ?val .
4   ?n local:successor ?succ . }
5                       => { ?n owl:imports  ?succ . }.
```

We measure the runtime of all three implementations for sets of 20, 40, 60, 80 and 100 number resources. For the approaches *LD-Fu* and *Cwm direct* the

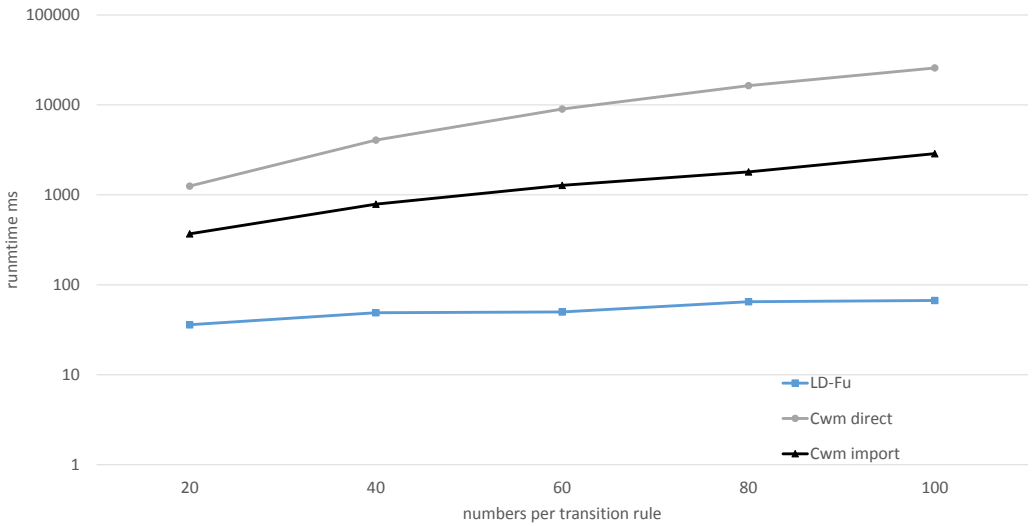


Figure 25: Average runtime from ten executions for different evaluation set-ups to retrieve and modify one set of number resources.

interaction ends when the last number in a set does not refer to a next successor to retrieve, i.e. the fixpoint is reached and in case of *LD-Fu* all manipulations are done. For *Cwm import* we had to decide manually how often the rule is applied and thus how many numbers are retrieved and when the interaction stops. The results are shown in Table 8 and Figure 25. We provide the average runtimes from ten executions to reduce variations.

Table 8: Average runtime from ten executions for different evaluation set-ups to retrieve and modify one set of number resources.

number set size	LD-Fu	Cwm direct	Cwm import
20	36 ms	1254 ms	369 ms
40	49 ms	4063 ms	790 ms
60	50 ms	9017 ms	1276 ms
80	65 ms	16383 ms	1801 ms
100	67 ms	25770 ms	2876 ms

LD-Fu is able to execute the interaction by orders of magnitude faster than the other two approaches with Cwm. Also the growth-rate of the runtime with the increasing amount of number resources is much lower with Data-Fu compared to the Cwm approaches. Specifically, *LD-Fu* requires 1.9 times longer for the interaction with 100 number resources compared to the interaction with 20 resources, where *Cwm direct* and *Cwm import* require 20.6 and 7.9 times longer respectively. The *LD-Fu* achieves this time advantage, even though it additionally includes manipulations, by leveraging the streaming execution model: With *LD-Fu* newly retrieved triples are simply added to the TripleQueue for processing in the operator plan, where further lookups and manipulation requests are done on-the-fly. Cwm has to apply the rules repeatedly over the increasing dataset in its triple store.

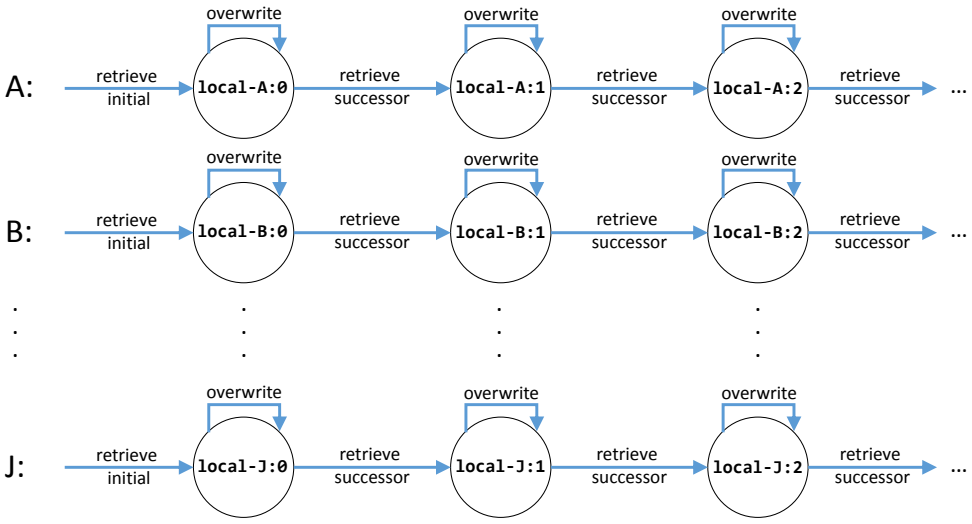


Figure 26: Interactions with ten sets number resources.

To evaluate the capabilities of our system with regard to an increasing amount of rules we execute the same interaction of retrieving and overwriting successors of numbers again, with ten different sets of numbers (A-J) in parallel. The different sets of numbers are distinguished by different namespaces. Each of the three evaluation set-ups employs ten rules for the retrieval each addressing another namespace (and ten additional transition rules in case of *LD-Fu*)[81], analog to the previously shown rules in Listings 13–15. Figure 26 illustrates the implemented interaction with the ten number resource sets.

The results for the different implementations are shown in Table 9 and Figure 27 as average from ten runs. Again, *LD-Fu* executes the interaction by order of magnitudes faster with a lower growth rate than Cwm, even though *LD-Fu* includes the manipulation requests. Specifically, *LD-Fu* requires 1.6 times longer for the interaction with 100 number resources compared to the interaction with 20 resources, where *Cwm direct* and *Cwm import* require 43 and 11.6 times longer respectively. In the case of the most interactions (10×100) *Cwm direct* requires over 13.5 minutes and *Cwm import* over 25 seconds, the Data-Fu engine handles the same interactions in 0.22 seconds.

Comparing the results of the interactions with a single number resource set and the interactions with ten number resource sets we see that the *LD-Fu* suffers less than Cwm from the ten times increased workload or the larger amount of employed rules: On average for the individual sizes of number sets

- *LD-Fu* requires 2.4 times longer,
- *Cwm direct* requires 23.9 times longer,
- *Cwm import* requires 7 times longer,

[81] The same interaction and manipulation of all ten number resource sets could also be achieved with only two rules addressing all number sets in *Ld-Fu*.

Table 9: Average runtime from ten executions for different evaluation set-ups to retrieve and modify ten sets of number resources.

number set size	Data-Fu	Cwm direct	Cwm import
20	140 ms	18810 ms	2212 ms
40	165 ms	85668 ms	5519 ms
60	178 ms	251503 ms	10416 ms
80	186 ms	478476 ms	17600 ms
100	224 ms	810204 ms	25666 ms

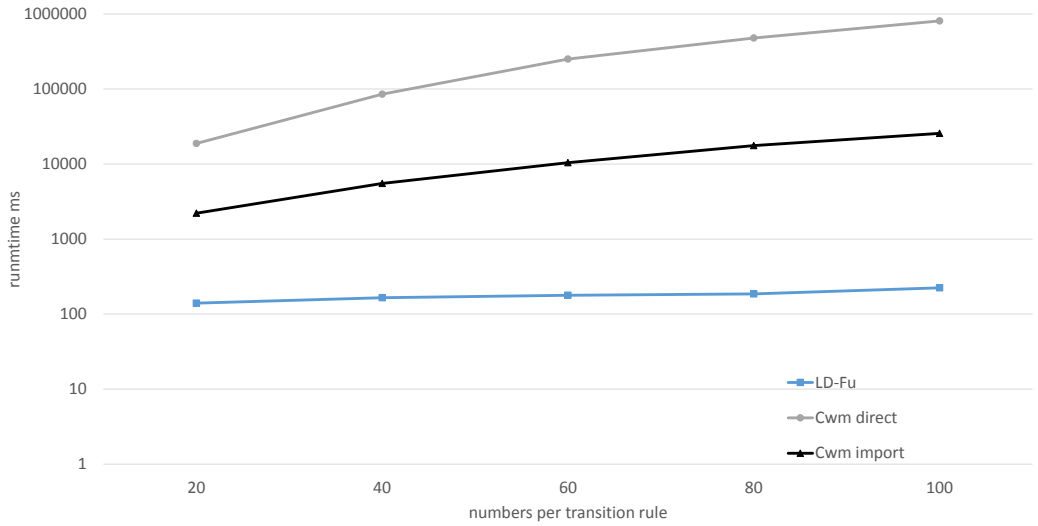


Figure 27: Average runtime from ten executions for different evaluation set-ups to retrieve and modify ten sets of number resources.

when running with ten rules over ten number sets compared to one single rule over one number set.

The reason for this time advantage is the parallel processing of our execution mode, which allows to retrieve resources from all sets in parallel combined with data processing. As we execute the manipulating requests in parallel as well, the introduction of transition rules does not impede the scalability of our system.

Following the results of the comparison with Cwm, we devise an additional evaluation setting to test the runtime of our system when performing larger amounts of manipulations. Similar to the previous evaluation setting, we retrieve number resources that are identified during runtime as successor of an already found number. We fix the size of the number sets to 1000, i.e., we deploy sets of 1000 consecutive number resources that are distinguished with their namespace. Then we retrieve and overwrite the numbers of every set with a respective request and transition rule, i.e., there are two rules per number set.

We evaluate the runtime of our system with 20, 40, 60, 80 and 100 number resource sets, thus performing between 20 000 and 100 000 lookups and manipulations. Additionally, we measure the time needed to establish the physical operation plan (including creation of the logical operator plan and optimisation)

separately to compare it with the total execution time. The results are shown in Table 10 and Figure 28.

Table 10: Average execution and planning time from ten executions of a LD-Fu program to interact and manipulate sets of 1000 resources.

number sets	execution time	evaluation plan
20	5932 ms	9 ms
40	11840 ms	11 ms
60	17564 ms	15 ms
80	24385 ms	24 ms
100	32133 ms	27 ms

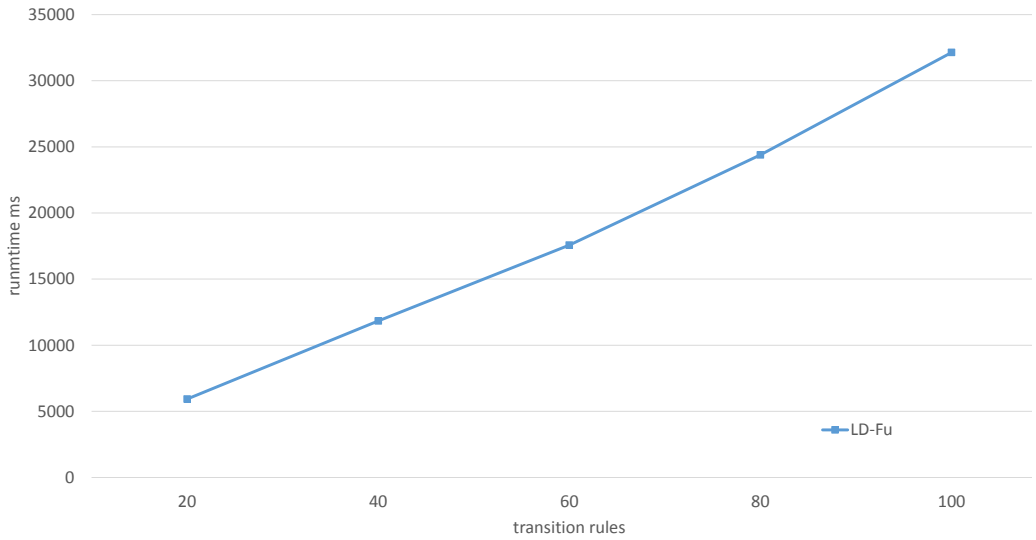
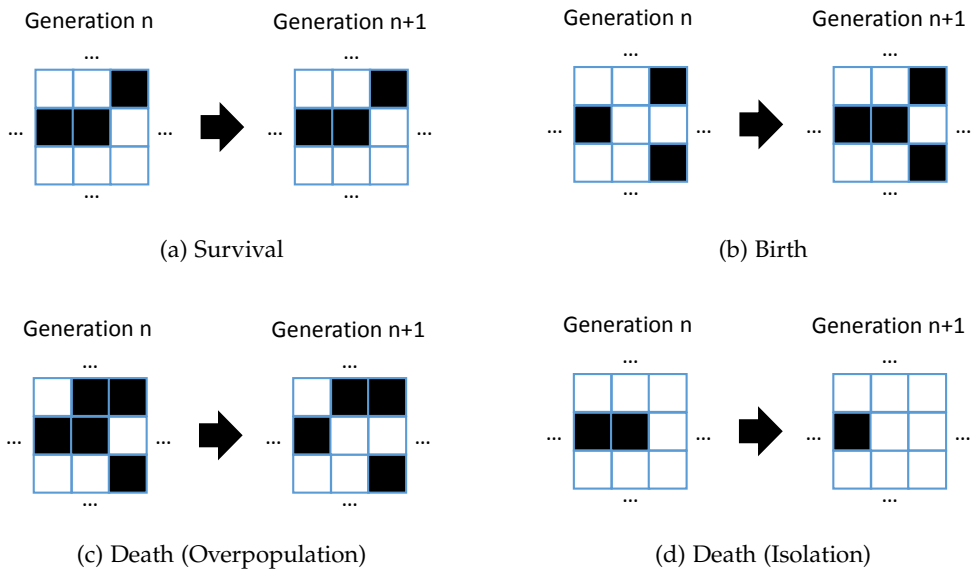


Figure 28: Average execution and planning time from ten executions of a LD-Fu program to interact and manipulate sets of 1000 resources.

The results of the evaluation for large amounts of interactions show that our system scales well up to thousands of manipulations even on commodity hardware. The system retrieved and manipulated 100 000 resources in about 0.5 min. The necessary time required to establish the evaluation plan increases with the number of rules, but remains a very small fraction of the overall runtime and is therefore negligible.

Overall, the important result of the experiments in this section is not the absolute runtime advantage of LD-Fu programs over Cwm, but the relatively better scalability in the face of an increasing amount of resources to interact with and rules in the programs. The evaluation shows the advantages of the parallel processing are not negatively influenced by the presence of transition rules and the necessity to perform manipulating requests. The integration of transition rules in the operator plan allows the development of applications manipulating Web resources dynamically, while preserving a low runtime requirements behaviour as outlined in Section 3.5.



(a) Survival (b) Birth

(c) Death (Overpopulation) (d) Death (Isolation)

Figure 29: Examples of the evolution of a cell (center) according to Game of Life laws.

4.5.2 *Game of Life*

In this section we evaluate the runtime of a repeatedly executed LD-Fu program that includes resource manipulations. For this purpose we implement Conway's Game of Life [36] with a LD-Fu program.

Game of Life is a zero-player game, i.e., it evolves independently without any manual intervention as a result of the start configuration of the game board. Specifically, the game board is a two dimensional grid of square cells. Thus, every cell has eight neighbours (two neighbours on the horizontal, vertical, and the two diagonal axis respectively). In particular, the neighbours of cells on the outer boarder of the board are the cells on the opposite side of the board. E.g., the right neighbor of the cell in the first row on the right outer boarder is the left outer cell on the first row. Every cell can be in of two states: alive or dead. A population of cells on the board evolves from generation to generation according to the following laws, which are illustrated in Figure 29:

SURVIVAL Every cell with two or three alive neighbouring cells survives for the next generation.

DEATH Every alive cell with four or more alive neighbouring cells dies due to overpopulation. Every alive cell with one or no alive neighbours dies from isolation.

BIRTH Every dead cell with exactly three alive neighbours comes to life for the next generation.

The evolution laws are applied to all cells simultaneously to advance the population from one generation to another.

For our experiment we implement a game board as Linked API, where every cell is a URI-identified resource. In particular, we use the row and column coordinates of the cells in the identifier. The resources are deployed locally and we use

namespace prefix `gol` for the deployed cells. The cell resources can be retrieved (via GET) and changed (via POST). If a cell is retrieved the returned RDF graph contains ten triples as shown in Listing 16, providing information about the state of the cell, the amount of its live neighours[82], and links to all its neighours.

Listing 16: Example RD graph of a Game of Life cell.

```
gol:r2c4 gol:state "alive" .
gol:r2c4 gol:livingNeighbors "4" .
gol:r2c4 gol:leftUpperNeighbour gol:r1c3 .
gol:r2c4 gol:leftNeighbour gol:r2c3 .
gol:r2c4 gol:leftLowerNeighbour gol:r3c3 .
gol:r2c4 gol:upperNeighbour gol:r1c4 .
gol:r2c4 gol:lowerNeighbour gol:r3c4 .
gol:r2c4 gol:rightUpperNeighbour gol:r1c5 .
gol:r2c4 gol:rightNeighbour gol:r2c5 .
gol:r2c4 gol:rightLowerNeighbour gol:r3c5 .
```

We use an LD-Fu program to implement the logic of the evolution on the game board. In particular we use

- request rules to retrieve all cell resources,
- deduction rules to derive if a cell needs to change its state,
- transition rules to execute derived changes.

We use a deduction rule for every possible state of a resource in combination with their neighboring cells that lead to a change in the state of the resource. Thus the deduction rules encode the Game of Life laws regarding *birth* and *death*. As cells that do not fall in the categories of *birth* and *death* do not need to be changed, i.e., cells that stay dead or alive in the next generation, we do not require additional rules to encode *survival*.

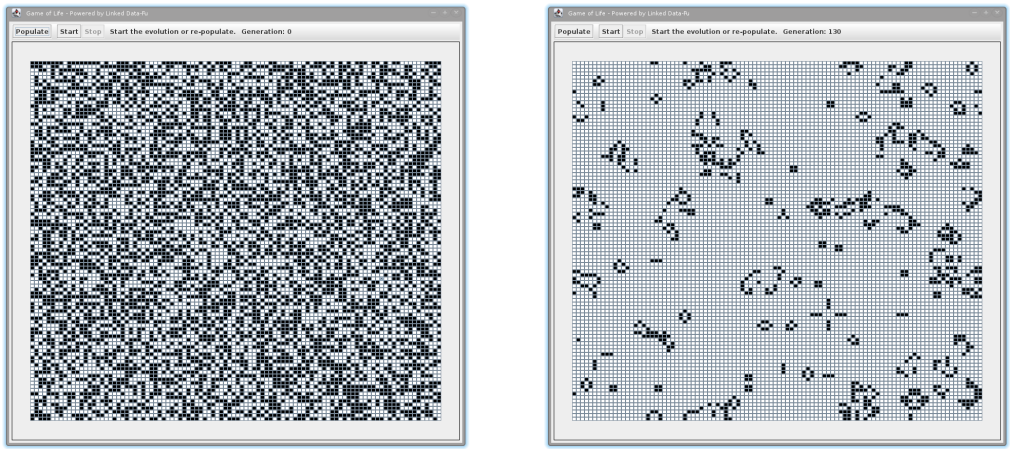
Listing 17 shows the deduction rule and transition that encode the *birth* law in our Game of Life implementation. Similarly we require further deduction rules to encode the *death* law. The complete list of all employed rules to encode Game of Life is provided in Appendix A.3 in Listing 20.

Listing 17: Example rules to encode the birth law in Game of Life.

```
#(1) Derive birth
{ ?cell  gol:state "dead" .
 ?cell  gol:livingNeighbors "3" . } => { ?cell gol:change "birth" . }.

#(2) Execute birth
{ ?cell gol:change "birth" . }
                => { [] http:mthd httpm:POST ;
                        http:requestURI ?n ;
                        http:body { ?cell  gol:state "alive" . } .  }.
```

[82] In our implementation the Linked API provides the number of live neighbours directly, rather than having the client application counting live neighbours, as arithmetic expressions are not supported by the semantics of LD-Fu.



(a) Generation 0 (b) Generation 130

Figure 30: Visualisation of the population (black) on a Game of Life board.

It is noteworthy, that dedicated deduction rules are not strictly necessary. Every deduction rule could be directly encoded as transition rule. E.g., in Listing 17, the body of deduction rule (1) could be used as body of the transition rule (2), thus the transition rule would directly encode the *birth* law. However, for our evaluation we intentionally want to use an LD-Fu program that contains all three kinds of rules.

We initialise the game board randomly, where every cell has a 50% chance to be alive in generation 0. We execute the LD-Fu program repeatedly, where every execution is equivalent to one evolution step from one generation to the next. Since every resource has to be retrieved, the amount of processed data is the same at every evolution step, independently of the population size. Specifically, we use the *blocking multi* threading model with fourTripleWorker and 100 RequestWorker (see Section 3.4). The individual threads are kept alive between different executions (see Section 3.5.3).

The LD-Fu program is inherently deterministic, as the Game of Life laws result in a unique decision about the change of every cell. We use the *random* conflict resolution approach. However, we ensure that no cell resource is changed before it is retrieved to guarantee that every new generation is calculated correctly, as if all laws are applied simultaneously.

Additionally we visualise the current state of the resources with the Java Swing architecture[83]. Figure 30 shows the game board after initialisation in generation 0 and after 130 generations.

Different areas of a game board decrease in population size until they reach stable or osculating configurations, other configurations continuously add new live cells to the board [36]. We let random initialised boards of size 10×10 (i.e., 100 cell resources), 33×33 (i.e., 1 089 cell resources), and 100×100 (i.e., 10 000 cell resources) evolve for 100 generations and measure the required time to derive and execute the necessary changes. Figure 31 shows the necessary time for

[83] http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html; retrieved 2015-04-10.

an evolution step together with the population size and the necessary manipulation requests for every generation. A detailed list of results can be found in Appendix A.2 in Tables 32 to 34.

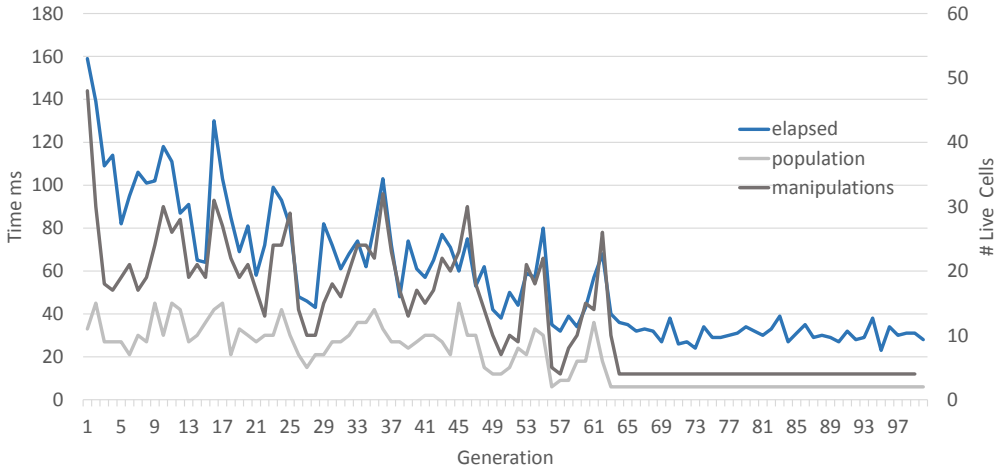
With a game board of size 10×10 the average time to transition from one generation to the next was 58 ms, thus on average our system executed the LD-Fu program with a frequency of about 17.2 Hz. With the relatively small size of the board Figure 31(a) shows particularly how the amount of manipulations influences the necessary runtime. After 64 generations the population stabilised and the evolution started to oscillate between two configurations, with only two manipulating requests. Once the population stabilised the execution frequency also became relatively stable with about 32.4 Hz (standard deviation of 3.7).

With a game board of size 33×33 the average time to transition from one generation to the next was 138 ms, thus on average our system executed the LD-Fu program with a frequency of about 7.2 Hz. Figure 31(b) shows how with a decreasing population the necessary manipulations decline and in turn the runtime decreases. E.g., over the last 10 generations the average execution frequency was about 35.8 Hz (standard deviation of 3.3) with on average 55 manipulating requests. The fact that the repeated execution with 55 requests results in roughly the same frequency as the repeated execution with only 2 requests on the game board with size 10×10 points to the fact that with a small amount of manipulating requests the derivation of the required changes via the deduction rules dominate the required runtime and not the requests themselves. Thus, we see evidence that the manipulating requests (i.e., I/O-bound tasks) are effectively executed in parallel with the data processing (i.e., CPU-bound tasks).

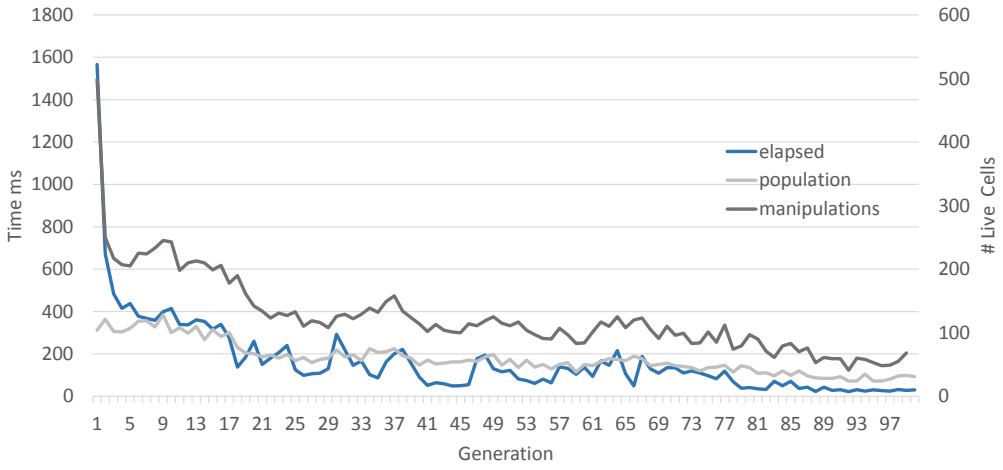
In early generations with a higher population count we see that the runtime decreases faster than the number of required manipulating requests: E.g., in the first ten generations the amount of executed requests decrease by 51% (from 499 to 243 requests), however the necessary runtime decreases by 74% (from 1566 ms to 414 ms). Again, the increasing relative runtime reduction is a result of the parallel execution of requests and data processing. As the amount of processed data (from all 1084 cell resources) is the same at every evolution step, while the amount of necessary requests decrease. With a decreasing amount of requests, the dominating factor of the overall runtime changes from I/O-bound tasks to the CPU-bound tasks.

With a game board of size 100×100 the average time to transition from one generation to the next was 592 ms, thus on average our system executed the LD-Fu program with a frequency of about 1.7 Hz. In large populations the effect of an increasing reduction of runtime relative to the amount of requests (while the amount of processed data stays the same between generations) becomes more apparent (Figure 31(c)). In the first ten generations the amount of executed requests decrease by 56% (from 4478 to 1808 requests), however the necessary runtime decreases by 89% (from 7774 ms to 641 ms).

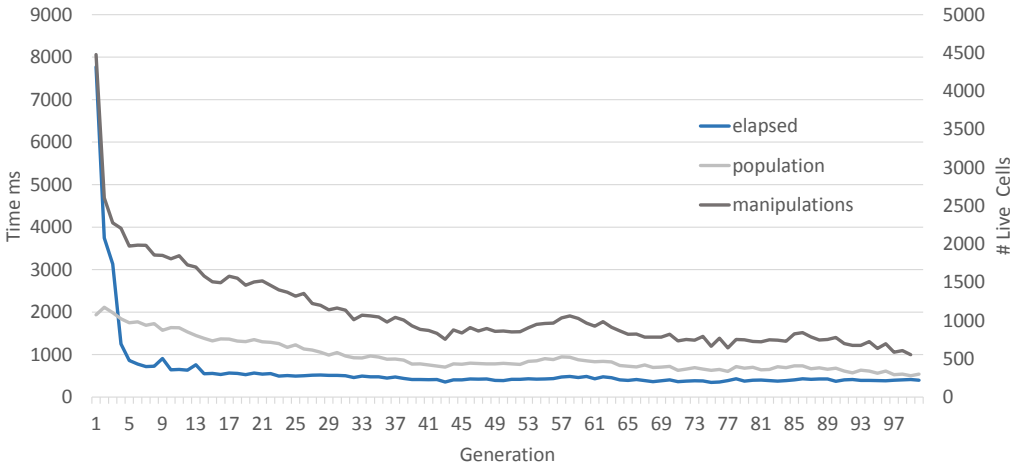
In summary, the experiments provide evidence that the manipulating requests as I/O-bound tasks integrate well in the execution model of our system, as the requests are executed in parallel with the data processing, thus reducing the effect of the manipulation of remote resources on the overall runtime. Specifically, we see that the repeated retrieval of resources, data processing and subsequent ma-



(a) Size 10x10



(b) Size 33x33



(c) Size 100x100

Figure 31: Runtime to derive and execute evolution changes in a Game of Life implemented with LD-Fu.

nipulation can be performed with high frequency. The frequency of the repeated execution is scalable over an increasing amount resources. Thus applications to monitor or iteratively manipulate Web resources with high frequencies can be realised.

4.6 RELATED WORK

Pautasso introduces an extension to BPEL [81] for a composition of REST and operation-oriented Web services. REST services are wrapped in WSDL descriptions to allow for a BPEL composition. Our approach focuses on a native composition of resource-oriented services, rather than shifting the focus on operations. A comparison between REST services and operation-oriented services is given in [83].

There exist several approaches that extend the operation oriented WS-* stack with semantic capabilities by leveraging ontologies and rule-based descriptions (e.g., [114, 27, 21]) to achieve an increased degree of automation in high level tasks, such as service discovery, composition and mediation. Those approaches extending WS-* are known as Semantic Web Services (SWS). An Approach to combine REST services with SWS technologies (in particular WSMO-Lite [122]) was investigated by Kopecky et al. [60]. In contrast to SWS, REST architectures do not allow to define arbitrary functions, but are constrained to a defined set of methods and are built around another kind of abstraction, i.e., resource. Therefore our approach is more focused on resource and data-centric scenarios in distributed environments such as the Web.

Active XML introduces service calls as XML nodes that are placeholders for new XML documents that can be retrieved from the service [1]. The service calls are comparable to the traversal of hypermedia links between resources. An active XML document corresponds to the result graph of an LD-Fu program. In contrast to Active XML, LD-Fu programs discover links to new resources rather than links to function calls. The resource model of Linked APIs allows more flexibility for the API design and evolution, as a client can generally try to execute any HTTP method on a discovered resource, whereas the Active XML equivalent would be constrained to the given operation in the original link.

The scripting language S [14] allows to develop Web resources with a focus on performance due to parallelisation of calculations. Resources can make use of other resources in descriptions, thus enabling a way of composing REST services. S does not explicitly address the dynamicity of REST APIs, as it has no explicit facilities to leverage hypermedia controls or to infer required operations from resource states.

RESTdesc [120] is an approach in which REST Linked Data resources are described in N3-Notation. The composition of resources is based on an N3 reasoner and stipulates manual interventions of users to decide which hypermedia controls should be followed.

Hernandez et al. [50] proposes a model for semantically enabled REST services as a combination of pi-calculus [75] and approaches to triple space computing [26] pioneered by the Linda system [37]. They argue, that the resource states can be seen as triple spaces, where during an interaction triple spaces can be

created and destroyed as proposed in an extension of triple space computing by Simperl et al. [99]. Our interaction model is in contrast not based on an explicit choreography of service calls, but is more focused on data-driven decisions with regard to interaction and manipulation. Similar to the idea of triple spaces is the composition of REST resources in a process space, proposed by Krummenacher et al. [61] based on resources described using graph patterns.

Stream reasoning systems from the area of complex event processing (e.g., [4] and [90]) are capable to deduce implicit information and derive decisions from on-the-fly processed data, similar to LD-Fu programs. However, these stream reasoning systems employ timestamps for retrieved data items and infer information over sliding windows over the incoming data. In contrast, our approach has a dedicated notion of a current state of the resources upon which an application is built, independently from the size of the data, which constitutes such a state. To constrain the size of the processed data of a state we employ the ability to define in a fine-grained manner what resource to take into account as well as what reasoning constructs to use.

Finally, production rule systems also employ rule-based reasoning to derive necessary actions from the current state of the world to achieve a goal. The current state of the world is reflected in a database as working memory, similar to the result graph of an LD-Fu program. Well-known examples of production rule systems are Drools[84] and Jess[85] [51]. Drools is part of the Knowledge is Everything tool suite of the Red Hat JBoss Middleware[86] portfolio and implements forward and backward chaining reasoning built upon the RETE algorithm [31]. Jess is a programming language and engine for declarative rules; it employs backward chaining inferencing to reason over and manipulate Java objects. In contrast to the production rule systems described above, our approach is directly built upon the Web, as it allows for dynamic link traversal and allows for the on-the-fly consideration of distributed and interlinked schemata. However, our system can be classified as a production rule system for Web.

4.7 SUMMARY AND FUTURE WORK

In this chapter we focused on applications that go beyond data retrieval and processing by also allowing for the manipulation of Web resources. In particular we describe how Linked Data can be combined with REST –a core Web technology– and applications can be defined upon such a combination, which preserve the capability to achieve low runtimes while allowing for the dynamic change of Web resources.

Both Linked Data and REST are concerned with interlinked URI-identified Web resources. While Linked Data is focused on a uniform data model and the provision of interlinked schemata, REST addresses an interaction modal that allows retrieval as well as manipulation of resources with a constrained set of methods. Thus, we outlined the synergies from the combination of both technologies, which relate to the creation of Linked API architectures that enable

[84] http://www.drools.org/; retrieved 2015-04-10.
[85] http://www.jessrules.com/; retrieved 2015-04-10.
[86] http://www.jboss.org/; retrieved 2015-04-10.

an interaction with little entropy. Finally, we provided a formal definition as state transition system for the interaction with Linked Data resources according to the principles of REST, thus positively answering research question 2.1. A Linked API is defined via the set of resources it exposes, in such a way that an API adhering to the standard specifications for HTTP and the Linked Data principles is compatible with the state transition interaction model. Additionally, we described how existing REST APIs on the Web can be wrapped to also provide a Linked Data interface.

We extended the previously introduced linked programs with transition rules to LD-Fu programs, which allow to derive manipulating requests from a given RDF graph. Thus, desired manipulations can be described where the links to the effected resources can be derived out of the current state of the state transition system. Further, potential changes can be conditioned to the state of retrieved resources as reflected in the state transition system. Via transition rules derived requests represent a transition between the states in the state transition model. Consequently, our approach to describe desired manipulations of a program with transition rules provides a solution for research question 2.2.

An LD-Fu program retrieves resources with request rules and derives new information according to deduction rules. The resulting graph forms the basis for the deduction of necessary requests according to transition rules. We answered research question 2.3 by describing the execution semantics of an LD-Fu program as a reaction to the information available about the current state of a state transition system. The available information is represented in the result graph of a program. Thus, an execution of an LD-Fu program implies the calculation of the result graph and the subsequent (i.e., after arrival at the fixpoint) construction of manipulating requests out of result bindings from the result graph. In particular, we described the potentially non-deterministic behaviour of an LD-Fu program with respect to the effected changes of resources and outlined different approaches to mitigate the resulting uncertainty. Further, we described the repeated execution of a program to achieve an iterative progression through the defined state transition system, i.e., the manipulation of resources. While the calculation of a result graph always reaches a fixpoint given a finite set of resources, the repeated execution might exhibit different bahaviours, which might not be a priori predictable.

Finally, we conducted experiments to provide evidence that the introduced approach to manipulate resources does not impede the runtime of applications. Specifically, we have shown that programs with transition rules scale with respect to the amount of manipulated resources. Further, we established that applications operating with high frequency repetitions of LD-Fu executions are possible.

Consequently, we can confirm Hypothesis 2, as we have shown how desired manipulations of applications can be expressed in the context of declarative rule-based programs, which are built upon the combination of Linked Data and REST in the form of state transition systems. Resources that are modified can be dynamically discovered via link traversal in existing resources. The execution of manipulating requests can be defined over conditions on the state of resources,

which are evaluated with performance characteristics that allow for applications with low runtimes.

Future work building upon this chapter relates to the increase of expressiveness of the employed rules to improve the possibilities for the definition of application logic. For instance, disjunctive graph patterns in rule bodies can allow to formulate alternative conditions under which a manipulating request is derived, while the conjunctive BGPs require to define an individual rule for every alternative.

Furthermore, the possibility to include arithmetic expressions in LD-Fu programs would result a in more expressive semantic allowing for a wide range of additional application scenarios to be implemented. However, an increase of semantic expressiveness comes at the price of increased computational costs. Additionally, a rise in complexity can also result in undesired characteristics, e.g., the inclusion of arithmetic expressions in a rule based program might lead to the creation of new identifiers, and thus inherently preventing the existence of a fixpoint even without an interaction with remote resources. The identification of the right balance between complexity and expressiveness for Web-based application remains an important challenge.

Finally, the combination of process modeling approaches with the presented declarative rule-based approach can allow to incorporate explicit choreographies in the data-driven interaction with Web resources. Specifically, the inclusion of suitable choreography representations in the initial graph of an LD-Fu program in conjunction with the repeated execution of the program to iteratively execute a defined process can be explored. Establishing a bridge between explicit process definitions and declarative rule programs might result in a better control over the execution while retaining dynamic reactions and fostering parallel execution.

So far we focused on the specification of the desired interaction and manipulation of client applications with Web resources. However, developing a Web-based application involves other high-level tasks, like the identification of suitable resources to start an interaction. Additionally, resources might impose constraints on the interaction with regard to the communicated data. An application requires certainty about such constraints to establish an interaction to achieve objectives. In the following chapter we address description mechanisms for Web resources that allow applications to acquire necessary information and identify resources adequate to achieve the application goals.

5

WEB RESOURCE SEARCH

5.1 INTRODUCTION

The interaction with Linked Data Web resources is based on the traversal of links. However, an application that follows links requires a starting point. Such a starting point are the first resources the application can retrieve to obtain links to further resources. In the context of REST these first resources of an interaction are called *entry resources* [124]. For a LD-Fu program entry resources are directly contained in the set of initial resources R or can be derived from links in the initial graph G.

The identification of suitable entry resources of APIs for an application, i.e., resource search, is a high level task in the context of interacting with Web resources. Resource descriptions can help to provide a higher level of automation for such tasks. Other examples of high level tasks in the context of interaction with Web resources that profit from description mechanisms include composition, comparison, and clustering of services [20].

In this chapter, based on our work in [106, 110], we focus on graph pattern-based descriptions of Linked Data resources to support the interaction with those resources by applications. In particular we describe how graph-pattern-based descriptions can be used for the task of resource search, which consists of

- matching a resource search request with resource descriptions,
- ranking available resources according to the matching degree with a request.

Many approaches [61, 103, 120, 121] propose the use of graph patterns for the description of Linked Data-based APIs. Here, we do not introduce a further alternative approach, but describe the applicability of BGP-based descriptions to support applications in the interaction with Web resources. Thus, we focus on the underlying commonality of the existing approaches applied to the introduced state transition interaction model, specifically for the high level task of resource search.

> ✎ **Example 29**
>
> To further improve sales ACME introduces two concert recommendation systems in their Linked API. Both recommendation systems are represented by Linked Data Web resources, to which users can send messages

via POST requests to receive recommendations about upcoming concerts in their area:

- `acme:simpleRecommender` allows to send the identifiers of a band and a location. The response contains a recommendation for a concert of a similar band in the given location.
- `acme:complexRecommender` allows to send the identifier of a band and precise longitude and latitude coordinates. The response contains a recommendation for a concert of a similar band nearby the provided coordinate point as well as the ticket price for the concert.

The recommendation resources are expected to serve as entry resources for applications, where users can look for concerts and subsequently buy tickets. To foster the use of the new recommendation systems ACME wants to provide descriptions for the two resources, so application developers can search for the resources in a precise manner. Specifically, developers should be able to identify, which of the two recommendation resources suits their needs, with respect to required input data and expected information returned.

Since ACME expects to further extend their Linked API, entry resources should be searchable in an automated fashion.

Other work [84], which introduces the name *Linked Services*, proposes the idea that service descriptions should also be exposed as Linked Data. The original approach, pursued in the SOA4All project[87] and leading to the iServe service repository[88], uses the Minimal Service Model (MSM)[89] as a lightweight RDF model of Web services. MSM-based descriptions characterise input and output expectations of service operations by annotating structural message parts with ontology classes.

Similarly OWL-S[90], an ontology for the description of Semantic Web Services [21, 27, 114], hinges on the annotation of syntactical messages (usually assumed to be bourne in plain XML) with ontological classes. E.g., the input as well as the output of the recommendation systems of Example 29 could be annotated with `acme:Artist`, among other classes. Other information on how input or output maps to ontological concepts is contained in non-semantic transformation instructions for lifting and lowering, e.g., with XSL Transformations [56]. Additional information are given with pre- and post-conditions in rule expressions, e.g., with the Semantic Web Rule Language (SWRL)[91], which describe requirements and effected changes of a service invocation.

However, annotations with ontological concepts do not clarify the relation between the input and corresponding output. E.g., it would remain unclear in Example 29, if the `acme:Artist` in the response is the same as in the input, or –as it is the case– refers to a different band. Additionally, the use of RDF (and BGPs)

[87] http://www.soa4all.eu/; retrieved 2015-04-10.
[88] http://iserve.kmi.open.ac.uk/; retrieved 2015-04-10.
[89] http://iserve.kmi.open.ac.uk/ns/msm/msm-2014-09-03.html; retrieved 2015-04-10.
[90] http://www.w3.org/Submission/OWL-S/; retrieved 2015-04-10.
[91] http://www.w3.org/Submission/SWRL/; retrieved 2015-04-10.

inherently allows to declare the class membership of instances, which makes additional annotations cumbersome.

The Web Service Modeling Ontology (WSMO)[92] improves on class-based annotations by allowing to define explicit choreographies with the Web Service Modeling Language (WSML)[93]. Such a choreography allows to indicate not just which classes (called concepts in WSML) are communicable, but also enables a description of relations (an n-ary generalisation of binary RDF property relations). However, the relations do not clarify the relation between input and output. Thus, in Example 29 it remains undefined if the recommended concert relates to the artist provided in the request or to a different artist.

The approaches to the use of graph patterns for service descriptions can be traced back to an extension of OWL-S, in the work described in [95]. Here, the graph patterns were used to encode pre- and post-conditions. We focus on descriptions for the required input and expected output of an interaction with a Web resource. The presented algorithms for the search of resources can, however, be applied analogously for graph patterns describing pre- and post-conditions.

The achieved changes with respect to the current state (see section 4.3) of a successful request are often inherently defined by the applied HTTP method with the exception of POST (see Section 5.2).

5.1.1 *Challenges*

We address the following challenges in the context of runtime requirements and entropy (see Section 1.1.1):

- To resolve the uncertainty about what resources can serve an application as entry resources requires to identify, if the necessary data is available to start an interaction, and if the desired information can be retrieved from an interaction. BGP descriptions are mostly focused on the graph representation of resources, in particular the communicated data to retrieve or influence the graph representation. However, as the interaction with resources depends on the applied operations, so does a description providing necessary information to identify entry resources. Therefore, to establish useful descriptions an interpretation with respect to the employed interaction model has to be defined.

- When searching for resources it can be too limiting to just consider resources whose description perfectly match the search conditions. Resources that nearly meet the defined requirements can be worth considering, especially if no perfect match can be found. Thus, we need to rank resources according to the degree a resource matches a search request without discarding non-perfect matches.

- Although the search for entry resources of an application can be considered part of the development process, runtime requirements are still a challenge to be addressed. In particular, if many search requests have to be evaluated

[92] http://www.w3.org/Submission/WSMO/
[93] http://www.w3.org/Submission/WSML/

against a multitude of different resource descriptions, the response time of the system has to remain tolerable. Additionally, if additional resources and their descriptions are added, existing search requests should be updated with new results, without having to evaluate the complete search again. Consequently, an approach to match and rank resources against search requests has to be scalable with respect to the considered descriptions and committed search requests.

5.1.2 *Contributions*

Our contributions are as follows:

- We describe how BGPs can be used to describe input data of requests and output data of responses. In particular, we define the relationship of the descriptions to the interaction model described in Section 4.3. Thus, we define the semantics of the descriptions with respect to the HTTP methods applied in an interaction. (Section 5.2) We also show how BGPs can be used to define interaction templates, which can be used to formulate search requests for identifying resources. (Section 5.3)
- We define several metrics to describe to what degree a given interaction template in a search request matches a resources description. In particular, we define two metrics based on the used vocabulary terms in templates and descriptions, which can be used as preliminary heuristic for matching and ranking. Further, we provide a metric based on the actual containment of triple patterns in the BGPs of description and template, which allows to precisely determine, if a resource matches a given search request and identify differences. (Section 5.4)
- We describe an architecture to match and rank BGP-based resource descriptions with interaction templates. Specifically the architecture also updates existing search requests, when new resource descriptions are committed to the system. In particular, we show how service search is divided into sub-tasks, which can be leveraged in a distributed computing architecture. Based on an algorithm to quickly prune non-matching triple patterns, the proposed architecture achieves scalability with respect to the amount of descriptions and search requests. (Section 5.5)

5.2 GRAPH PATTERN DESCRIPTIONS

In a REST-based interaction with Linked Data resources only the HTTP methods can be applied to the resources. The semantics of the HTTP methods itself is defined by the IETF[94] and does not need to be explicitly described. However, the input payload of a request with a specific method can be subject to conditions with respect to the used vocabulary and the information that have to be included in the payload. Furthermore, the output payload in a response depends on the used method and the information given in the request payload.

[94] https://tools.ietf.org/html/rfc7230 et seq.; retrieved 2015-04-10.

We use BGPs to describe input and output of a resource for a given HTTP method.

➳ Definition 18: BGP Resource Description

A BGP resource description is a tuple (ipd, opd) consisting of

- a BGP ipd as input description,
- a BGP opd as output description.

For a URI-identified resource $r \in \mathcal{U}$ and an HTTP method $m \in M$ the input description ipd represents a constraint of possible input payloads D^{req} in requests (m, r, D^{req}), such that there is exactly one result binding for the BGP ipd from D^{req} for all variables in ipd, i.e.,

$$\exists! \mu \in \Omega_{D^{req}}(ipd) \wedge dom(\mu) = \mathcal{V}_{ipd}$$

For a URI-identified resource $r \in \mathcal{U}$ and an HTTP method $m \in M$ the output description opd represents a guarantee of output payload D^{res} in responses to requests $\beth(m, r, D^{req}) = (c, D^{res})$ for any D^{req} adhering to the input description, such that there is exactly one result binding for the BGP opd from D^{res} for all variables in opd, i.e.,

$$\exists! \mu \in \Omega_{D^{res}}(opd) \wedge dom(\mu) = \mathcal{V}_{opd}$$

Further, to encode the relation between input and output it has to hold true, that for all result bindings for the input description ipd from the request payload D^{req}, there is a result binding for the output description opd from the response payload D^{res}, that maps the variables that appear in ipd as well as in odp to the same term.

$$\forall v \in \mathcal{V}_{ipd} \cap \mathcal{V}_{opd} \forall \mu_i \in \Omega_{D^{req}}(ipd) \exists \mu_o \in \Omega_{D^{res}}(opd) : \mu_i(v) = \mu_o(v)$$

A BGP resource description details for a resource the requirements on the input data for an interaction, as well as the guarantee for the output data, with respect to the employed HTTP method. In the following we detail the BGP descriptions for the different HTTP methods with respect to the state transition system interaction model[95] as definied in Section 4.3.

- A GET request retrieves the RDF graph representation of the addressed resource and does not change the current state:

  $$\delta^s(\sigma_k, (GET, r_i, D^{req})) = \sigma_k$$

  $$\delta^o(\sigma_k, (GET, r_i, D^{req})) = (2xx, \overline{r_i^k})$$

  Consequently, a resources can be considered to ignore any input payload in the request, i.e., the payload in the request can be considered empty,

[95] It would be analogously possible to use BGPs to define requirements on the state σ_k at which an interaction can be performed as pre-condition, and guarantees on the resulting state $\delta^s(\sigma_k, req)$ as post-condition.

i.e., $D^{req} = \emptyset$. Thus, the input BGP description also is empty, i.e., $ipd = \emptyset$. However, other approaches consider parameters, encoded as key-value pairs to be described with an input graph pattern description (cf. [103]). As the response of a GET request returns the state of the addressed resource r_i, the output description odp also reflects the possible states of r_i, i.e.,

$$\exists \mu \in \Omega_{\overline{r_i^k}}(opd) \wedge dom(\mu) = \mathcal{V}_{opd}$$

- A DELETE request removes the addressed resource.

  $$\delta^s(\sigma_k, (DELETE, r_i, D^{req})) = \sigma_k \setminus \{\overline{r_i^k}\}$$

  $$\delta^o(\sigma_k, (DELETE, r_i, D^{req})) = (2xx, \emptyset)$$

  The removal of a resource does not require an input payload (i.e., $D^{req} = \emptyset$) and yields an empty output payload (i.e., $D^{res} = \emptyset$). Therefore, input and output payload are completely determined by the DELETE method and no further BGP description is required, as it is in every case $(\emptyset, \emptyset)$.

- A PUT request overwrites a resource or creates a new resource.

  $$\delta^s(\sigma_k, (PUT, r_i, D^{req})) = (\sigma_k \setminus \{\overline{r_i^k}\}) \cup D^{req}$$

  $$\delta^o(\sigma_k, (PUT, r_i, D^{req})) = (2xx, \emptyset)$$

  The RDF graph representing the addressed resource r_i after a successful PUT request is the graph in the input payload. Thus, the input BGP description also reflects the possible state to which r_i can be changed, i.e.,

  $$\exists \mu \in \Omega_{\overline{r_i^{k+1}}}(opd) \wedge dom(\mu) = \mathcal{V}_{opd}$$

  As the output payload of the response is always empty (i.e., $D^{res} = \emptyset$), so is is the output description (i.e., $opd = \emptyset$).

- A POST request can have various effects on the current state and result in different output payloads in the response.

  $$\delta^s(\sigma_k, (POST, r_i, D^{req})) = post_i^s(\sigma_k, D^{req})$$

  $$\delta^o(\sigma_k, (POST, r_i, D^{req})) = (2xx, post_i^o(\sigma_k, D^{req})$$

  If data is submitted in the request payload D^{req} to a data- handling process, the output payload in the response D^{res} is the result of some computation carried out over D^{req}. The relation between input and output is given as the common variables in input and output description are mapped to the same terms. As interactions with the POST method are the only requests that require both a non-empty input and non-empty output description, the relation established via shared variables is specifically important.

✎ **Example 30**

The recommendation resources of ACME allow to submit data via POST. Listing 18 shows the input and output BGP descriptions for the simple and complex recommendation resources of ACME. For both resources the variable `?artist` appears in input and output description, signifying that both are bound by the same term, i.e., the identifier of a band. The output description shows that the recommended concert (`?event`) is performed by a band that is similar to the band given in the request payload. Thus, the relationship between input and output is expressed.

After a successful POST request to one of the recommendation resources, a new resource is created representing the recommendation. The variable `?i` in the output descriptions will be bound by the identifier of the created resource, which can be used to lookup the recommendation again[a].

[a] It would be equivalently possible that the POST request to recommendation resources does not create a new resource and just returns the response, in which case ?i would be bound by a blank node.

Listing 18: BGP descriptions for recommendation resources.

```
# input pattern description for simple recommendation resource
_:i p:performer ?artist.
?artist rdf:type p:Artist.
_:i p:location ?town.
?town rdf:type schema:City.

# output pattern description for simple recommendation resource
?i rdf:type p:recommendation.
?i p:concert ?event.
?event p:performer ?p.
?p p:similar ?artist.
?event rdf:type p:Event.


# input pattern description for complex recommendation resource
?rec p:performer ?artist.
?artist rdf:type p:Artist.
?rec geo:nearby _:p.
_:p rdf:type geo:Point
_:p geo:latitude ?lat.
_:p geo:longitude ?long.

# output pattern description for complex recommendation resource
?i rdf:type p:recommendation.
?i p:concert ?event.
?event p:performer ?p.
?p p:similar ?artist.
?event rdf:type p:Event.
?event p:price ?p.
?p rdf:type p:Cost
```

5.3 MATCHING

To enable the search for resources over their BGP descriptions, we define resource templates, which can be used to formulate a search request.

➳ Definition 19: BGP Resource Template

A BGP resource template is a tuple (ipt, opt) consisting of

- a BGP ipt as input template,
- a BGP opt as output template.

The input template ipt represents all possible input RDF graphs an application can provide in the payload D^{req} of a request by specifying that the application can provide input payloads such that ipt has at least one result binding from D^{res} for the variables in ipt, i.e.,

$$\exists \mu \in \Omega_{D^{req}}(ipt) \wedge dom(\mu) = \mathcal{V}_{ipt}$$

The output template ipt represents the output RDF graphs such an application expects to be delivered in the payload D^{res} of a response by specifying that the output payload has to result in at least one result binding of opt from D^{res} for the variables in opt, i.e.,

$$\exists \mu \in \Omega_{D^{res}}(opt) \wedge dom(\mu) = \mathcal{V}_{opt}$$

Further, it has to hold true, that for all result bindings for the input template ipt from the provided request payload D^{req}, there is a result binding for the output template opt from the response payload D^{res}, that maps the variables that appear in ipt as well as in opt to the same term.

$$\forall v \in \mathcal{V}_{ipt} \cap \mathcal{V}_{opt} \forall \mu_i \in \Omega_{D^{req}}(ipt) \exists \mu_o \in \Omega_{D^{res}}(opt) : \mu_i(v) = \mu_o(v)$$

The BGP resource templates follow the same syntax as the resource descriptions. Similar to the resource description a resource template encapsulates all necessary request information needed to formalise an agents possibilities and wishes, including the expected relation between the input and output.

Therefore, the question of whether a given resource description matches a resource template correlates to the problem of graph pattern containment. The input graph pattern of a resource description must be contained in the resource template's input pattern. This containment relation implies that every graph that satisfies the template input graph pattern must also satisfy the service description's input graph pattern. Intuitively, this is to say that the input an application can provide satisfies the requirements for the input payload of a request to the described resource. Specifically, it is also possible for the input template to specify additional data, which the application can provide for an input payload, even though a matching resource does not require the data.

✎ **Example 31**

An application can express in an input template that it can provide information about a location and a music genre. If an input description details that the input payload of a request requires just the information about the location, template and description are matching, since the description is contained in the template. However, if a description details that information about location and an artist is required, template and description are not matching.

Matching the output graph patterns works in an analogous way. The output graph pattern of a resource description contains the output graph pattern of a template, which implies that every graph that satisfies the resource description's output graph pattern also satisfies the template output graph pattern. So the required containment relation of the output patterns is dual to that of the input graph patterns. Intuitively, again this means a service output has to provide enough information to satisfy the request, but can provide more.

✎ **Example 32**

An application can express in an output template that it is looking for information about a concert. If a service details in an output description that it provides information about concerts and genres, template and description are matching, since the template is contained in the description. However, if template details it requires information about concerts and ticket prices, template and description are not matching.

A BGP O is contained in another BGP P, if O is a sub-graph pattern of P:

➳ **Definition 20: Sub-graph Pattern**

Let O and P be two BGPs. O is a sub-graph pattern of P, denoted as $O \dot{\subseteq} P$, iff for all possible RDF graphs G and every result binding μ_O for O from G there is a result binding μ_P for P from G, such that for every variable v_o in O there is a varaible v_p in P, where μ_O maps v_o to the same term as μ_p maps v_p, i.e.,

$$\forall G \forall \mu_O \in \Omega_G(O) \exists \mu_P \in \Omega_G(P) \forall v_o \in \mathcal{V}_O \exists v_p \in \mathcal{V}_P : \mu_O(v_O) = \mu_P(v_P)$$

The desired relations between the patterns in a resource description $(\mathtt{ipd}, \mathtt{opd})$ and a resource template $(\mathtt{ipt}, \mathtt{opt})$ is summarised as follows:

$$\mathtt{ipd} \dot{\subseteq} \mathtt{ipt}$$

$$\mathtt{opt} \dot{\subseteq} \mathtt{opd}$$

5.4 RANKING

The matching based on graph pattern containment subsists in two binary decisions (one for the input and one for the output), answering if a resource de-

scription completely matches a resource template. However, it is sensible to assume that often resources only partly satisfy the requirements of an application, or that an application has not all the necessary data for the invocation. Therefore, resources should be ordered according to the degree they match to a given search request. To allow for a flexible search approach, we allow for the ranking of resource descriptions against resource templates by providing continuously-valued matching metrics:

PREDICATE SUBSET RATIO (psr) measures to what degree the set of predicates used in one pattern is subsumed within the set of predicates in another pattern.

RESOURCE SUBSET RATIO (rsr) measures to what degree the set of named resources, in subject or object position, used in one pattern is subsumed within the set of named resources in another pattern.

CONTAINMENT RATIO (cr) measures to what degree triple patterns in one graph pattern are contained in the other pattern.

In the following we describe the metrics in detail and provide an example on how they are applied to resource descriptions and templates.

5.4.1 *Containment-based metric*

The metric *cr* measures to what degree a BGP P_{sub} is contained in another pattern P_{super}.

➳ Definition 21: Containment Ratio

For two BGPs P_{sub} and P_{super} the containment ratio *cr* for P_{sub} in P_{super} is the fraction of triple patterns in the largest subset of P_{sub}, which forms a sub-graph pattern of P_{super}, i.e.,

$$cr = max\left(\frac{card(T)}{card(P_{sub})}\right) \text{ with } T \subseteq P_{sub} \wedge T \dot{\subseteq} P_{super}$$

where $card(BGP)$ denotes the number of triple patterns in a graph pattern BGP

Intuitively, the containment ratio measures how many triple patterns have to be removed from a BGP, to make it a sub-graph pattern of another BGP.

The calculation of the containment *cr* is based on the power set $2^{P_{sub}}$ of triple patterns derived from the graph pattern P_{sub}. The largest set of triple patterns $T_{max} \in 2^{P_{sub}}$ (i.e., the set with the most triple patterns) in the power set is identified, that is still contained in P_{super}:

$$T_{max} \dot{\subseteq} P_{super}$$

T_{max} is not necessarily unique in $2^{P_{sub}}$. However, just one of the largest triple pattern sets that are contained in P_{super} needs to be identified for the calculation of *cr*.

The metric *cr* describes the ratio between the number of triple patterns in the identified set T_{max} and the overall number of triple patterns in the original pattern P_{sub}, from which the power set is derived. *cr* measures precisely to what degree a graph pattern is subsumed by another, thus expressing the containment degree of one pattern in relation to another.

However, a naive calculation of *cr* would require to check for every set $T_i \in 2^{P_{sub}}$), if it is contained in the other pattern P_{super}, which can be computationally expensive. To mitigate this problem we propose an algorithm that prunes triple patterns as early as possible, if they are not contained in P_{super}. Further, the algorithm avoids an unnecessary testing of elements in $2^{P_{sub}}$. Since the algorithm is based on an iterative extension of subsets of P_{sub} an identified set $T_k \in 2^{P_{sub}}$), that is not contained in P_{super} can be disregarded for further extension. This exclusion is possible due to the monotonicity of the containment relation: Adding a triple pattern to an already not contained set cannot result in a larger contained set.

$$T_k \not\subseteq P_{super} \Rightarrow \forall tp_i \in P_{sup} : T_k \cup \{tp_i\} \not\subseteq P_{super}$$

To improve the readability of the Algorithm 4, we define two functions: Let Q be a BGP and G an RDF graph:

- A function that substitutes all variables in Q with URIs of the same name, thus mapping the pattern to a graph:

  $$skolem(Q) = \{\langle rn(s), rn(p), rn(o)\rangle | \ \langle s, p, o\rangle \in Q\}$$

  The function rn defines the substitution of the variables:

  $$rn(x) = \begin{cases} x & \text{if } x \in \mathcal{U} \cup \mathcal{L} \cup \mathcal{B} \\ :x \quad \text{with } :x \in \mathcal{U} & \text{if } x \in \mathcal{V} \end{cases}$$

- A function that returns $true$ if a given BGP has a non-empty result binding from G and $false$ otherwise.

  $$match : (G, Q) = \begin{cases} true & \text{if } \exists \mu \in \Omega_G(Q) \wedge dom(\mu) = \mathcal{V}_Q \\ false & \text{otherwise} \end{cases}$$

Algorithm 4: Determine pattern containment ratio

Input : $P_{sub} = \{tp_{sub-1}, ..., tp_{sub-n}\}$; a BGB with triple patterns tp_i
Input : $P_{super} = \{tp_{super-1}, ..., tp_{super-m}\}$; a BGB with triple patterns tp_i
Output: cr; containment ratio of P_{sub} for P_{super}

```
1  G ← skolem( P_super)
2  MatchingTriplePattern ← ∅
3  MatchingGraphPattern ← ∅
4  ResultCandidates ← ∅
5  foreach tp_i ∈ P_sub do
6      if match(G,{tp_i}) = true then
7          add tp_i to MatchingTriplePattern
8          RP_i ← {tp_i}
9          add RP_i to MatchingGraphPattern
10 while MatchingGraphPattern ≠ ∅ do
11     ResultCandidates ← MatchingGraphPattern
12     MatchingGraphPattern ← ∅
13     foreach RP_i ∈ ResultCandidates do
14         foreach tp_k ∈ (MatchingTriplePattern − {RP_i}) do
15             if match(G, RP_i ∪ {tp_k}) = true then
16                 add RP_i ∪ {tp_k} to MatchingGraphPattern
17 T ← one element in ResultCandidates
18 cr = card(T) / card(P_sub)
```

The algorithm takes the two graph pattern as input for which *cr* is to be calculated. In the initialisation phase the pattern P_{super} is skolemised (line 1) and the resulting graph G is matched with every triple pattern in P_{sub} individually (line 5). The matching triple patterns are stored in the set MatchingTriplePattern (line 7). Further, MatchingGraphPattern is initialised with a set of graph patterns that match the graph G (line 9), where each of these initial graph patterns contain exactly one of the matching triple patterns.

The algorithm keeps running as long as additional larger graph patterns are found that are contained in P_{super}, i.e., that have a result binding from G (lines 10–16). The outer `for` loop in the execution phase iterates over the (largest) found matching graph patterns in ResultCandidates (line 13). In the inner `for` loop (line 14) the algorithm extends the considered graph pattern with one of the matching triple patterns. If the resulting larger graph pattern matches G, the pattern is stored (lines 15–16). The algorithm terminates when no larger graph patterns that match G can be found (line 16).

In the worst case with respect to the runtime of Algorithm 4 the analysed graph patterns of template and service discription are equivalent. In such a case with $cr = 1.0$ the calculation time is equivalent to the calculation with a naive approach since all sets in $2^{P_{sub}}$ are evaluated. However, for the scenario of resource search it can be expected, that the amount of templates and descriptions to compare is quite high, where the majority of template and description pairs have a very low matching degree or do not match at all. Under these circumstances

the proposed algorithm provides significant advantages with its pruning capabilities: In the best case with respect to the runtime the analysed graph patterns do not match ($cr = 0.0$). In this case the algorithm would identify the individual triple patterns $tp_i \in P_{sub}$ as not matching with P_{super} and do no further calculations at all.

To apply the *cr* metric to the search of resource descriptions given a resource template, the containment degree has to be calculated for the input pattern and output pattern. In particular, it has to be calculated to what degree the input pattern of the resource description is contained in the input pattern of the resource template. Furthermore, it has to be calculated to what degree the output pattern in the resource template is contained in the output pattern of the resource description. The resulting values (cr_{input} and cr_{output}) provide an insight how well a resource description matches a given resource template in terms of pattern containment, rather than just relying on a binary matching decision.

5.4.2 *Vocabulary-based metric*

Intuitively the metrics *psr* and *rsr* indicate to what degree a resource description and a resource template are using the same vocabulary. These vocabulary-based metrics allow to test whether a description and a template use some of the same named resources and predicates (and to what degree). Therefore the vocabulary-based metrics provide a mechanism to discover resources, which are close to a given template, but are not necessarily completely matching.

➳ Definition 22: Predicate Subset Ratio

For two BGPs P_{sub} and P_{super} the predicate subset ratio *psr* for P_{sub} in P_{super} is the fraction of predicates in P_{sub} that are also contained in P_{super}.
Let $Pred_{sub} = \{p | \langle s, p, o\rangle \in P_{sub}\}$ the set of predicates in P_{sub}.
Let $Pred_{super} = \{p | \langle s, p, o\rangle \in P_{super}\}$ the set of predicates in P_{super}.

$$psr = \frac{card(Pred_{sub} \cap Pred_{super})}{card(Pred_{sub})}$$

➳ Definition 23: Resource Subset Ratio

For two BGPs P_{sub} and P_{super} the resource subset ratio *rsr* for P_{sub} in P_{super} is the fraction of resources in subject and object position in P_{sub} that are also contained in P_{super}.
Let $Re_{sub} = \{s | \langle s, p, o\rangle \in P_{sub} \wedge s \in \mathcal{U}\} \cup \{o | \langle s, p, o\rangle \in P_{sub} \wedge o \in \mathcal{U}\}$ the set of resources in P_{sub}.

Let $Re_{super} = \{s | \langle s, p, o\rangle \in P_{super} \wedge s \in \mathcal{U}\} \cup \{o | \langle s, p, o\rangle \in P_{super} \wedge o \in \mathcal{U}\}$ the set of resources in P_{super}.

$$rsr = \frac{card(Re_{sub} \cap Re_{super})}{card(Re_{sub})}$$

Similarly to the pattern containment we have to distinguish between the metrics for input and output. A template input graph pattern can offer more data than actually needed by a described resource without endangering their compatibility. Therefore, the subset ratios for the input patterns have to measure, to what degree the named resources (respectively predicates) in the descriptions are used in the template. For the subset ratios of the output patterns the same concept applies in the opposite direction, because a described resource can offer more output data than required by the template. Thus, the subset ratios have to measure, to what degree the named resources (respectively predicates) in the template are used in the description. In particular, for a resource description (ipd, opd) and a resource template (ipt, opt) the vocabulary based metrics are given by

$$psr_{input} = \frac{card(\text{predicates in } ipt \cap \text{predicates in } ipd)}{card(\text{predicates in } ipd)}$$

$$psr_{output} = \frac{card(\text{predicates in } opt \cap \text{predicates in } opd)}{card(\text{predicates in } opt)}$$

$$rsr_{input} = \frac{card(\text{resources in } ipt \cap \text{resources in } ipd)}{card(\text{resources in } ipd)}$$

$$rsr_{output} = \frac{card(\text{resources in } opt \cap \text{resources in } opd)}{card(\text{resources in } opt)}$$

Note, that if a graph pattern is completely contained in another pattern (i.e., *cr*=1.0), the subset ratios must necessarily result in a metric of 1.0.

The vocabulary-based metrics do not regard the structure of the graph patterns to match resources descriptions and templates. The metrics *rsr* and *psr* are therefore less precise discovery metrics compared to *cr*, i.e., even if all vocabulary-based subset ratios result in a value of 1.0, it is not guaranteed, that resource description and template match in terms of pattern containment. On the other hand, if *cr*=1.0 for the input and output patterns, a complete match is guaranteed. However, the vocabulary-based metrics are less computationally expensive to calculate. To achieve a scalable discovery system *rsr* and *psr* can be used to filter service descriptions for a given template: If either *rsr* or *psr* result in a value of 0.0, it can be inferred without additional calculation that the value of *cr* has to be 0.0. Furthermore, other low values for *rsr* and *psr* can be used as thresholds: If one of the vocabulary-based metrics falls below the defined threshold, the calculation of *cr* can be omitted. The setting of this threshold depends on the required reaction time of the resource search system, as well as the desired values for precision and recall.

✎ **Example 33**

An application developer is looking for a concert recommendation service for a Web-based application. In particular, users are supposed to identify a band they like, as well as the location where the users want to go to a concert. The application subsequently should show concerts by bands that are similar to the provided band including the ticket prices for the concert. As the application is not specifically a mobile app, for which GPS might be available, the developer does not expect to have longitude and latitude information available.

The developer formulates a resource template and matches it against the descriptions of the two recommendation resources of ACME:

- `acme:simpleRecommender` expects a band and a location in the payload, and returns a concert of a similar band as recommendation, but no information on ticket prices.
- `acme:complexRecommender` expects a band as well as longitude and latitude data in the request payload, and returns a concert of a similar band including ticket prices as recommendation.

Thus the application has enough information to invoke the interaction with `acme:simpleRecommender`, which will not provide all the desired information, since no data about the price of concert tickets are delivered by `acme:simpleRecommender`. Interacting with `acme:complexRecommender` would provide all the desired information, but the application does not have enough data to enact the interaction; specifically the latitude and longitude coordinates of the location are missing.

Figure 32 shows how the template of the provider is matched against the recommendation resource description. Table 11 lists the metrics resulting from the matching of template and descriptions. It can be seen how the *psr* and *rsr* metrics can serve as an estimation for the more precise *cr* metric.

In the given example no perfect match for the template can be found, i.e., neither of the recommendation resources matches in input as well as in output. Thus the application developer has to decide:

RESOLUTION 1 Choose `acme:simpleRecommender` and forgo some of the desired results or acquire them from another Linked API.

RESOLUTION 2 Choose `acme:complexRecommender` and try to acquire missing information for the request, e.g., with the help of another Linked API, which allows to resolve locations to latitude and longitude coordinates.

In terms of ranking with regard to the input `acme:simpleRecommender` would be preferred over `acme:complexRecommender`, but with regard to the output `acme:complexRecommender` would have to be preferred over `acme:simpleRecommender`. If a definite order of the resources is required, a weighted average can be calculated from the input and output metrics.

> The weights reflect which of the resolution strategies is preferred by the developer.

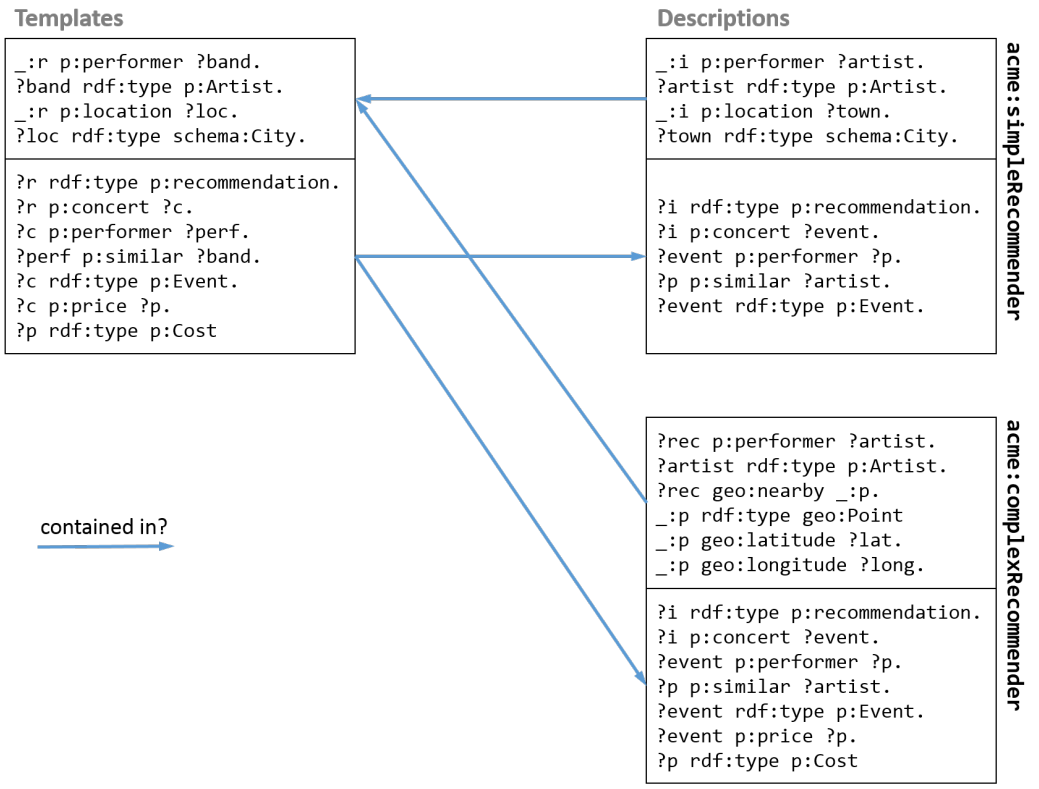


Figure 32: Search of recommendation resources by matching template with descriptions.

Table 11: Metrics for matching template with recommendation resource descriptions as shown in Figure 32.

	simple recommender		complex recommender	
	input	output	input	output
pattern containment	YES	NO	NO	YES
cr	1.0	0.71	0.33	1.0
psr	1.0	0.8	0.4	1.0
rsr	1.0	0.66	0.5	1.0

5.5 SEARCH ARCHITECTURE

In this section we propose an architecture for systems to search for Web resources, where the described metrics can be calculated. In particular, our architecture comprises a resource description and resource template repository in a cloud-based environment, where descriptions as well as templates can be submitted, managed, and the metrics for every combination of description and

template can be retrieved. The insight pursued in the architecture to achieve scalability is that the calculation of the metrics for every combination of description and template can be structured as a MapReduce problem [24].

Work already exists on the provision of REST-based repositories for service descriptions, and the design of such repositories according to Linked Data principles [79, 85]. We extend these approaches to resource descriptions as well as templates. Both description and template repository are identified by a URI. Users can send two BGPs in the payload of a POST request to both repositories as resource description or template respectively. When a description or template is submitted to a repository, a URI-identified resource is created containing both BGPs (i.e., the BGPs for input and output). In the standard REST manner, the resource can be retrieved by GET, updated by PUT with new BGPs, and it can be removed by a DELETE.

The architecture does not strictly adhere to the definition of Linked APIs, as the resources and communicated messages are BGPs rather then RDF graphs (cf. Definition 13). However, the BGPs can be embedded in RDF documents, to achieve a full adherence to the definition of Linked APIs. E.g., the BGPs can be part of more holistic service descriptions (like Linked USDL [86]), that detail business and operational aspects beyond the technical perspective of resource interaction as proposed in [110].

By managing templates as persistent resources in the same way as descriptions, matching against the descriptions becomes an on-going task. The described metrics are calculated in two directions, depending on the creation of new description or template resources:

- When a new resource template is uploaded and a resource is created, the template is matched against every existing resource description.
- When a new resource description is uploaded and a resource is created, every existing template is matched against the new description.

To search for an entry resource for a given task a template can be submitted. Every submitted template is stored and matched with all currently stored resource descriptions, i.e., the system checks for the containment of the input and output patterns and calculates the metrics Specifically, the input and output BGPs of the submitted template and all descriptions are parsed and the resource URIs in subject and object position are extracted as well as the URIs of all predicates. The sets of resource and predicate URIs are used to calculate the vocabulary-based metrics *rsr* and *psr*. All descriptions that do not reach an *rsr*– and *psr*–value above a pre-defined threshold are not further considered for matching, as these descriptions do not use the same vocabularies to a sufficient degree.

Subsequently the system calculates the metric *cr* for the template and the descriptions that resulted in vocabulary-based metric values above the threshold. In particular, the variables in the input template and the output descriptions are substituted with generated URIs, resulting in a template input and description output graphs. Thus, Algorithm 4 can be applied to calculate *cr* by evaluating

- subsets of the description input BGPs over the template input graph, and

- subsets of the template output BGP over the description output graphs.

Each set of metrics, generated in this way, for every combination of template and service descriptions is assigned a URI and stored as resource. Additionally, links from the template and description resources to the resource that contains the corresponding matching metric values are added. Thus, users can lookup the results for the template they submitted.

We employ an analogous process to allow resource descriptions to be updated, or to populate the system with new resource descriptions. A Service provider can submit (via POST request) resource descriptions, which are stored in the system and matched with all existing resource templates. The resulting metrics are also tagged with an identifier and complement the already existing results. Thus, every combination of template and service description has a set of results that is persistently saved and can be retrieved from the system.

If a search system is populated with descriptions for many resources, the amount of calculations for determining all the metrics can be quite high. However, the calculation of every individual metric for the comparison of a template with a resource description is a self-contained problem. This allows to perform the matching process in a parallelised fashion, where the calculation of a set of metrics for individual pairs of templates and descriptions are executed on different machines. Specifically, the matching of several BGPs can be described as a MapReduce process, which can be realised e.g., with Apache Hadoop[96], an open-source MapReduce implementation. To foster scalability the overall computation job of calculating all metrics of a submitted template (or description) for every existing description (template) is divided into smaller sub-tasks, which can be executed in parallel on different nodes in a cluster of machines (map function). Hadoop retrieves and combines the results of these sub-tasks to achieve the overall goal of the original computation job (reduce function).

For this purpose Hadoop implements a distributed file system, which spans over an arbitrary number of computers. Data is stored on blocks of this file system; these blocks are distributed randomly over all nodes in the cluster. Hadoop executes sub-tasks on the nodes that contain the blocks with the input data for the task. Therefore, an important consideration with any MapReduce problem is the locality of data; i.e., that the computation is reasonably well isolated from the communication of large amounts of data. Additionally, the blocks can be replicated several times to provide a safe mechanism against failure of nodes.

The resource descriptions, templates and matching results can be stored on the distributed storage. The system automatically calculates the matching metrics, when a template (or description) is submitted. Hadoop transfers and executes the code, that implements the matching mechanism, together with the submitted template (description) to the nodes where the resource descriptions (templates) are stored, rather than moving the data to the code. Hadoop tries to balance the workload of the nodes, taking into account that some nodes contain the same data blocks due to the described replication mechanism.

After calculation of the metrics, the map function assigns a time stamp and the identifier to every set of metrics and passes the generated results to the reduce

[96] http://hadoop.apache.org; retrieved 2015-04-10.

function. In our case the reduce function just gathers all the results and saves them persistently on the distributed file system. Since the proposed architecture also allows for updating templates and service descriptions by re-submitting a new version with the same identifier, we have to run a second house-keeping MapReduce job. This second job compares the newly generated results with the results that are already stored. If an existing description or template is updated with new BGPs, some of the generated results will also have the same identifiers. If results with the same identifiers are detected the older results are deleted, which can be checked by using the mentioned time stamps.

5.6 EXPERIMENTS

In the following we describe experiments to evaluate the scalability of our proposed architecture. Specifically, we evaluate the scalability of an implementation of the proposed search architecture and measure the scalability of the system with respect to employed computation nodes in a cluster of machines. Further, we confirm the capability of the algorithm to calculate the containment degree of two BGPs, to prune non matching patterns, which implies runtime advantages in scenarios where most matched BGPs are not expected to match.

All experiments are executed on the following hardware setup:

SETUP C A cluster of 11 virtual machines managed by the OpenNebula[97] toolkit. Each machine in the cluster operates with a singular virtual core based on an Intel Xeon E5520 processor at 2.27 GHz and 2 GB main memory. The operating system on each virtual machine is Ubuntu 10.04 LTS with 2.6.32 kernel.

We implement the search architecture on top of a Hadoop cluster (release version 1.0.0) connecting the virtual machines. This implies, that our cluster runs on top of a cloud, fully abstracting from actual physical hosts. The OpenNebula environment allows us to easily add and remove virtual machines as computing nodes to the cluster.

We develop a generator to create random pairs of related BGPs, that act as resource description or template in our system. The graph pattern pairs can be interpreted as service descriptions or templates, because both are syntactically equivalent. The generator allows for a precise control of the data used to evaluate the system. The BGPs are created within boundaries, set by parameters with respect to the following aspects of the BGPs:

- Minimum and maximum amount of triple patterns in a BGP.
- Probability that the same resources are used in the input and output pattern.
- Probability that the same predicates are used in the input and output pattern.
- Amount of variables present in the BGPs.

[97] http://www.opennebula.org; retrieved 2015-04-10.

- Probability that the same variables are used in the input and output pattern.

The concrete values for the parameters are chosen randomly between set boundaries with equal distribution individually for every generated BGP.

5.6.1 *Distributed Search*

To evaluate our distributed search architecture we generate 10 000 pairs of BGPs used as service descriptions. Both BGPs in these descriptions are composed out of a random number between 5 and 50 triple patterns. The resources in subject or object position, in the triple patterns for every respective pair are drawn out of a local resource pool consisting of 10 to 50 different URI-identified resources. These local resources are randomly drawn out of a global pool of 500 resources. The predicates in the triple patterns of the descriptions are also randomly drawn out of 3 to 25 different predicates in a local predicate pool. And again this local pool is randomly chosen out of a global pool of 250 predicates. So the difference between the local and global pools is, that the global pools of resources and predicates are used for all pairs of BGPs, whereas the local predicates and resources are only consistent for one pair of BGPs. Thus the size of the local pool determines the likelihood that the same URIs are used in input and output pattern of a description. This approach is chosen to establish a credible relationship between input and output and therefore results in credible datasets.

Additionally, the generator uses variables, rather than resources, in subject or object position with a probability of 0.3 in each case. A variable is used in predicate position with a probability of 0.2. For every BGP pair between 2 and 10 different variables are used. Since variables are already only locally valid within one description, no global variable pool to draw from is needed. Additionally, we generated a pair of graph patterns used as template with the same parameters.

We populate the search system with the resource description using different amounts of virtual machines, i.e., we use one, two, five, eight and ten worknodes in the Hadoop cluster, which are deployed on virtual machines of OpenNebula. With every configuration of nodes one additional virtual machine is needed to act as namenode for the Hadoop cluster. The namenode is used for the coordination of the distributed computation tasks, but does no computation itself.

The distributed HDFS storage is configured with a block size of 1MB with a replication of factor 3 for every used block. The 10 000 resource descriptions corresponded to 8.16MB of data and are therefore stored over 3 x 9 blocks on the cluster. We chose this setup to illustrate the capabilities of our architecture to leverage distributed computing to improve the performance of the system. However, in a real enterprise scenario, the settings of a discovery cloud could be easily adapted to the amount of templates and descriptions on the system. Our evaluation gives an insight on how to decide on such adaptions. The setup with only one worknode is equivalent to the processing in a non-distributed fashion and allows a comparison of the calculation on a stand-alone machine with the calculation in a distributed manner.

The matching process for the generated template over all service descriptions is triggered on every setup. We measure the execution time needed for the match-

Table 12: Overall execution time to calculate matching metrics and combine results for one template against 10 000 descriptions.

worknodes	execution	time (sec)	mean (sec)	std. dev. (sec)	std. err. (sec)
1	1.	477.3			
	2.	463.3	470.3	9.9	7.0
2	1.	283.7			
	2.	277	280.4	4.7	3.3
5	1.	169.7			
	2.	156	162.9	9.6	6.8
8	1.	155.3			
	2.	167.1	161.2	8.2	5.9
10	1.	134			
	2.	121.7	127.8	8.7	6.2

ing itself (i.e., first MapReduce job) and the overall execution time, which includes the time needed for the second MapReduce job to combine the newly calculated with the preexisting metric sets. To provide for comparable results regarding the overall time, we do not pre-populate the system with results. Therefore, the second MapReduce job uses every time only the 10 000 newly calculated metric sets as input (i.e., one for every combination of service description and template).

To account for fluctuations in network traffic we measured the matching on each setup twice. The results are shown in Table 12 and Table 13 with a graphical representation in Figure 33. The calculation of the metrics alone took between 81.4 sec, on ten worknodes, and 395.8 sec, on one worknode. The overall execution time was measured between 121.7 sec using ten worknodes, and 477.3 sec on one worknode. It can be seen, that the system scales well by adding additional worknodes between one to five nodes. Between five and eight nodes the execution time stagnates almost completely. By using ten worknodes in the Hadoop cluster the measured times are further decreased compared to the setup with eight worknodes, although the improvement is less significant than in the area between one and five worknodes.

The behavior between one and five worknodes is easily accounted for by the possibility to execute more computations simultaneously. The more nodes are on the system, the more sub-tasks can be launched at the same time. By employing up to eight worknodes no further decrease in execution time is achieved because of the ability of the Hadoop system to balance the workload with fewer nodes. The amount of blocks the input data fills on the HDFS storage limits the number of map tasks that can be executed. So in our case not more than nine map tasks can be launched (eight, and one with a small input size of 0.19MB). Therefore on settings with one to five worknodes almost every worknode has to execute at least two map tasks. This provides the namenode with more possibilities to distribute the map tasks among the worknodes.

Table 13: Execution time to calculate matching metrics for one template against 10 000 descriptions.

worknodes	execution	time (sec)	mean (sec)	std. dev. (sec)	std. err. (sec)
1	1.	394.3			
	2.	395.8	395	1	0.7
2	1.	223.6			
	2.	219.3	221.5	3	2.1
5	1.	120.6			
	2.	124	122.4	2.4	1.7
8	1.	121.7			
	2.	117.2	119.5	3.2	2.3
10	1.	81.4			
	2.	82.1	81.8	0.5	0.4

The load balancing takes into account that the worknodes contain non-disjunctive subsets of the overall input data set, due to the replication of blocks. The tasks can therefore be assigned in such a way that all employed worknodes contribute an equal amount of work to the calculation of the metrics. By using, for example, eight nodes, the coordinating namenode loses this possibility to some extent, since it always prefers parallel computation over load balancing (i.e., it will not wait for a worknode to finish if another worknode is already available). This also explains why a further, though diminishing, decrease of execution time is achieved by using more than eight nodes. In a setting with ten worknodes the namenode can (and must) decide not to use one of the nodes (and assigning the insignificant small map job to another). In this situation the least useful worknode is chosen to be disregarded by the namenode.

The effect of losing the possibility to balance the workload between the nodes can easily be avoided by choosing a smaller blocksize that allows for more map tasks than available nodes. But for our evaluation the choice of a larger block size allows the observation, that the improvement in terms of execution speed can not only be attributed to the increase of computation resources (i.e., adding additional CPUs and memory with every worknode), but also to the strategic distribution and execution of matching sub-tasks.

By comparing the results of the overall execution time and the matching time without housekeeping, similar observations can be made. The time needed to execute the second MapReduce job decreases due to the employment of a second worknode compared to a setup with only one node, but no further improvement can be achieved by adding additional nodes. The inputs for this second job are the calculated metrics, for every combination of service descriptions and template, which amounts to 2.68 MB of data. So only 3 MapJobs can be started simultaneously.

The standard deviation and standard error of the individual results are also shown in the tables, and represented in the figure (as bars). For the overall execution time the standard deviation ranges between 4.7 sec and 9.9 sec, which

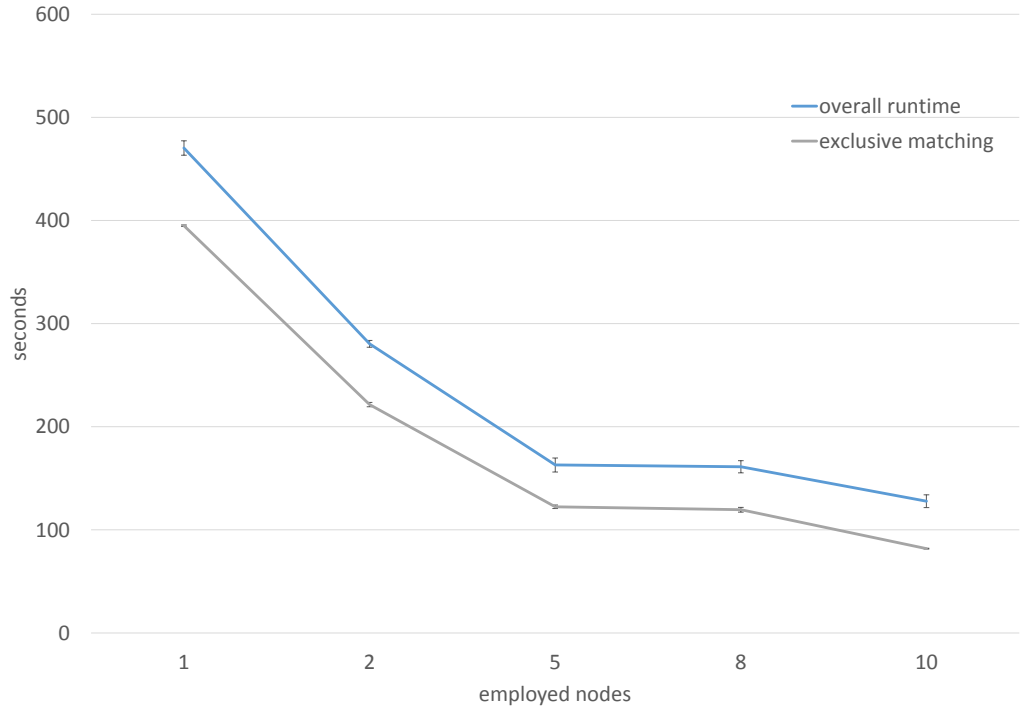


Figure 33: Graphical representation of execution time measurements

results in a standard error between 3.3 sec and 7 sec. For the exclusive matching process the standard deviation is measured between 0.5 sec and 3.2 sec, which results in a standard error between 0.4 sec and 2.3 sec. Those values clearly indicate the stability of the system.

Our results are not only valid for matching a template over resource descriptions, but also for populating the discovery system with a new resource description, because they are syntactically equivalent and the process to submit a new service description is symmetrical to the process of submitting a new template. In other words the 10 000 used graph pattern tuples could have just as well been interpreted as templates, that are already stored on the system and one new service description (i.e., the former template) is submitted.

5.6.2 *Containment Ratio Calculation*

To further evaluate our algorithm for the calculation of the *cr* metric we match a resource template with five sets each containing 1 000 resource descriptions on a single virtual machine. The parameters for the generation of every set of service descriptions are fixed except for the size of the global resource pool. In every service description the graph patterns contain 20 triple patterns for each input and output. Variables appear with a probability of 0.2 in every position of a triple pattern. The predicates in the triple patterns are randomly chosen from a local pool with 4 predicates drawn out of a global pool with 20 predicates. The local resource pool from which subject and object of the triple patterns are drawn contains 4 resources. The global resource pool for the individual sets of resource descriptions contains between 4 and 8 resources respectively.

Table 14: Measurements of runtime to calculate cr for a template against 1 000 descriptions.

resources in global pool	4	5	6	7	8
average cr value for input pattern	0.62	0.42	0.35	0.3	0.28
average cr value for output pattern	0.68	0.5	0.41	0.35	0.31
combined cr value (mean)	0.65	0.46	0.38	0.33	0.29
measured calculation time (sec)	85.7	51.1	42.9	35.7	32.6

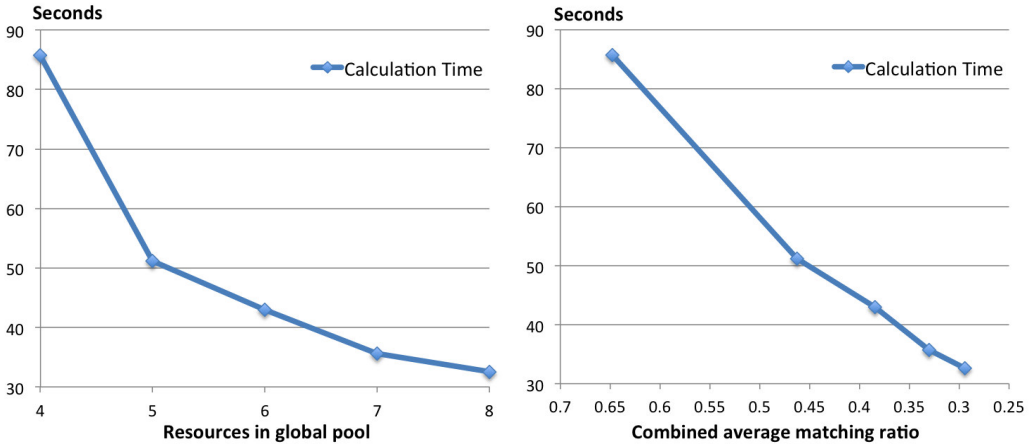


Figure 34: Graphical representation of measured runtime to calculate cr for a template against 1 000 descriptions.

The narrow margins for the boundaries of the pattern generation allow us to influence the expected matching degree represented by *cr*. The smaller the size of the global resource pool from which the descriptions are generated, the more descriptions have a high matching degree. We measure the necessary time to calculate all cr values (i.e. 1000 values for each input and output respectively) for each set of resource descriptions matched with the template.

The results shown in Table 14 show that the measured time to calculate the metrics ranges between 85.7 seconds and 32.6 seconds for the different service description sets. Table 14 shows the average *cr*-value for the different sets, generated with different sized global resource pools. An increase of the size of the global resource pool leads to a sub-proportional decrease of the average cr values. The necessary computation time decreases linear with a declining containment ratio. The decrease of calculation time can solely be attributed to the ability of the employed algorithm to prune not matching BGP subsets, since the patterns in the different service description sets are equal in size.

For a given resource search request (i.e., a resource template) we expect that most service descriptions in a repository will have low cr degrees. Intuitively this means, that very few resources will match the specific requirements formulated in a search request. Therefore, the ability of our algorithm to exclude not matching service descriptions fast is desirable for a scalable discovery system.

For the graphical representation of the results in Figure 34 we combined the average cr values of input and output to a mean value.

5.7 RELATED WORK

A whole strand of work revolves around graph matching in the context of ontology alignment and schema matching, e.g., [73]. The graph matching employed in these approaches is effectively similar to the graph pattern matching we use for resource search, although used for a different purpose. For a survey on graph matching with a focur on problem variation emerging from different domains see [34].

There are several discovery approaches that employ services descriptions based on description logic [40, 8, 67]. For OWL-S a discovery mechanism was proposed that uses OWL-S profiles to advertise services on a registry and to describe requests [115]. A matchmaker determines if a request matches the advertised services based on a scoring function, that classifies matches as exact, plugin, subsumed or failed. Li et al. [67] use a similar classification to determine matching degrees of service requests and service profiles. However, a discovery approach for services described with class-based annotations does not take into account the relationship between input and output messages of the communicated data.

Junghans et al. [55] argues that an intersection-based matchmaking of service requests and descriptions, based on the service functionality is insufficient, because it can not be guaranteed that a discovered service is able to satisfy the formalised request or that enough data is known to invoke the service. Further intersection-based matchmaking lacks the identification of the missing data. We addressed this problem by proposing a containment ratio, which can be used to identify the subset of data in a search request that matches. Thus, the missing information can be identified providing a clear understanding of necessary and potentially missing data with respect to the request.

Goal-driven approaches allow agents to formalize a goal with respect to the problem they try to solve [112, 57, 63]. Here a given goal has to be matched against the effects of invoked services. Stollberg et al. proposes to employ a hierarchy of goal templates [111] in the context of a state-based formal model of service descriptions to improve the scalability of the discovery process. The state-based model is similar to the communication of resource states in REST architectures, as used by Linked APIs. However, our approach focuses on the search of resources from which an application has to discover dynamically changing links, rather than statically identifying all services necessary to achieve a goal. Goal-driven approaches abstract from the details of service invocation by the definition of goals.

RESTdesc, another member of the Linked API family, proposes an interactive goal-driven discovery approach [120]. Here services are described with graph patterns in N3 rules. An agent starts with a high level plan that describes what he wants to achieve and discovers services during runtime by leveraging hypermedia links. However, mechanisms for an agent to decide on the degree of relevance of a found service to achieve the goals are not addressed.

5.8 SUMMARY AND FUTURE WORK

In this chapter we focused on the use of BGPs as descriptions of resources. The interaction and manipulation of Web resources is based on the traversal of links to dynamically discover resources at runtime in the retrieved representations of other resources. However, applications require resources, which serve as entry point to begin a series of interactions. BGP-based descriptions of resources can be used to identify the functionality provided by a Linked API and in turn allows to identify suitable entry resources.

Specifically, we described how BGPs can be used to describe the input and output payload of requests to interact with Web resources with respect to the applied HTTP method of the request, thus answering research question 3.1. In particular, POST requests require descriptions for input and output, where it is important to encode how the output is generated by calculations carried out over the input. We establish the relationship between input and output descriptions with shared variables in both BGPs.

Further, we described how search requests for resources can be formulated as resource templates, which also consist of BGPs. Templates can be used to encode the data available to an application to use in a request, as well as the the information sought after by the application. Similar to the resource descriptions, the templates express the relationship between potential input and desired output, with shared variables between BGPs. Templates can be matched with resource descriptions by verifying the graph pattern containment of the BGPs. Thus, it is possible to identify the resources that provide desired information, which can be requested using available information. Therefore, we have described how graph pattern-based descriptions can be used to search for resources with desired functionality and answered research question 3.2.

To allow for resource ranking beyond a binary matching decision for given search requests, we proposed two vocabulary-based metrics as well as a metric to measure the containment degree of two BGPs. The metrics allow to identify the resources that match a given template best, even if they do not match perfectly. The vocabulary-based metrics give a heuristic for the matching by describing if template and description use the same vocabularies. The containment metric describes to what degree a BGP is subsumed by another.

To allow for a scalable calculation of the metrics for large sets of templates and descriptions we provided an algorithm for computing the containment metric, which prunes non-matching results. Further, we described an architecture of a search system built upon distributed computing, which allows to divide the matching into sub-tasks executed in a cluster of machines. Thus, we answered research question 3.3 by describing how search results can be ranked in a scalable fashion, given graph pattern-based search requests. Finally, we conducted experiments, which provide evidence that the proposed architecture allows to match templates and descriptions in a scalable fashion in a distributed environment and confirmed the pruning capabilities of the algorithm to calculate the containment metric.

In summary we can confirm Hypothesis 3, as we have shown how graph patterns to describe the possible interactions and manipulations of Web resources

can be leveraged to allow for the scalable search of resources, which can serve as entry point for an interaction.

Future work related to the topic covered in this chapter can extend the expressiveness of the descriptions and search requests. Even as the actual state representations of resources remain at the level of BGPs, more expressive descriptions can allow to encode additional information about the resources. E.g., the fact that some triples only appear under certain circumstances in the RDF graph of a resource can be expressed by disjunctive graph patterns or optional triple patterns. Furthermore, arithmetic constraints on the values that are bound for the variables in the BGPs can establish more precise descriptions.

Analogously, templates can profit from more expressive descriptions by allowing to formulate more precise search requests. An increase in expressiveness of descriptions and templates requires an extended definition for the matching and ranking. Thus, an approach with more expressive descriptions and templates results in a more complex search process, which in turn can effect the required time to calculate matching degrees.

Another aspect of future work relates to the properties that are described and ultimately used to search for resources. We focused on the input and output payload when interacting with resources, which can be extended to the pre- and post-conditions before and after an interaction. However, other aspects can be relevant for the search of resources, like the costs for using the resources or potential service level agreements offered by a provider. Comprehensive search approaches can integrate such information regarding business and operational aspects with the technical information about the interaction with the resources.

Finally, the provision for inference and reasoning capabilities in the calculation of the metrics can also be the subject of future work. The containment of a pattern in another is not necessarily directly given, but could be entailed by schema information of the used vocabularies in the descriptions. An extended search approach could retrieve relevant schemata and logically derive matching descriptions for given templates, thus going beyond the direct BGP subsumption.

6

CONCLUSION

6.1 SUMMARY

We conclude by reiterating the hypotheses of this thesis and summarising our contributions to answer the associated research questions, before we outline potential future work related to the covered topics in Section 6.2.

> □ **Hypothesis 1**
>
> Declarative rule-based programs can be utilised to define desired dynamic retrieval and processing of Web resources for client applications in such a way that retrieval and processing can be executed in a highly parallel streaming fashion, thus enabling applications with short runtimes. (Chapter 3)

Research question 1.1 is concerned with the specification of desired dynamic interactions with Web resources based on on-the-fly discovered schemata. For this purpose we described rule-based programs that encode reasoning features intertwined with link traversal specifications. In particular, we introduced the notion of request rules, which infer required network lookups from the processed data. Request rules complement deduction rules, which infer and materialise implicit information from the processed data.

Research question 1.2 asks for the design of a data-driven parallel execution model for the interactions that is capable of on-the-fly processing of arriving data and schema information. To answer this question we introduced a parallel execution model for the evaluation of queries and rules, in which operators of a physical evaluation plan schedule each other in a data-driven manner. The scheduling of our parallel execution model avoids overhead from inter-process communication and thread scheduling of the operating system. At the same time the push approach caters to scenarios with dynamic network requests. Furthermore, we described several threading models to realise the execution model for the parallel evaluation of programs and identify the different trade-offs of the individual threading models. Finally, we described for each of the identified threading models how to probe for the termination of the evaluation, i.e., the identification of the processing fixpoint.

Research question 1.3 addresses the performance of the implementations of our execution model in the face of a intertwined data-processing and network lookups. We have shown that it is possible to separate the network-related workload from the processing-related workload. The separation of workload allows

to balance available computing resources between data processing and network lookups to minimise overall runtime depending on the application scenario.

In summary we confirm Hypothesis 1 insofar as we have described rule programs for Linked Data resources, which can be executed in a parallel scheduling execution model. Our experiments have confirmed that the proposed approach is capable of balancing the workload resulting from lookups and data processing. Although the network parameters unavoidably influence the execution behaviour our approach achieves short runtimes in relation to the time necessary to perform individual lookups.

☐ **Hypothesis 2**

The principles of Linked Open Data and Representational State Transfer can be combined to an interaction model for APIs based on state transitions that allows to design rule-based programs also for the manipulation of Web resources, which are dynamically discovered via link traversal, while preserving short runtimes. (Chapter 4)

Research question 2.1 is concerned with the combination of Linked Data and REST and the formalisation as a state transition system. First we identified discrepancies and synergies between Linked Data and REST. In particular, we described how the differences can be resolved and how both approaches can be combined to *Linked APIs*. The described combination defines a general architecture for APIs that overcomes drawbacks and realise advantageous of both approaches. Following the definition of Linked APIs we introduced a state transition interaction model for REST APIs as formal grounding for the manipulation of resources exposed in Linked APIs. Specifically, we defined the combined representation of Web resources as states, which allows to interpret manipulation upon the resources as transitions between states.

Research question 2.2 asks how manipulations of dynamically via link traversal discovered Web resources can be defined based on the state transition interaction model. We extend the previously introduced rule-based programs to allow for the specification of resource manipulations. Specifically, we introduce transition rules, which describe desired state transitions in the described model for Linked APIs. The manipulated resources as well as the messages to trigger state changes can be dynamically derived from the processed data, i.e., from retrieved resources. Thus, our approach allows for a definition of intended resource changes conditioned to the current state of resources leveraging reasoning capabilities to align heterogeneous vocabularies.

Research question 2.3 requires to define the execution semantics of the declarative rule-based program for the manipulation of Web resources. We characterised the potentially non-deterministic behaviour of the rule-based programs and introduce strategies to avoid or mitigate undesired effects. We further detailed the behaviour of the repeated evaluation of programs with resource manipulations and have shown how an iterative execution can be employed to achieve the goals of an application.

In summary we confirm Hypothesis 2. Although we cannot design autonomous agents that independently pursue given goals, we have shown how rule-based

programs that are defined over Linked Data resources adhering to REST principles can be used to define a dynamic application logic. Specifically, applications can be designed with such rule programs that react dynamically to the current state of Web resources and derive actions according to declarative rules to achieve defined states.

□ **Hypothesis 3**

Graph patterns to describe the possible interactions and manipulations of Web resources can be leveraged to allow for the scalable search of resources, which can serve as entry point for an interaction. (Chapter 5)

Research question 3.1 asks how potential interactions and manipulations of Web resources can be described with graph patterns with respect to applied operations. We described how BGPs can be used to describe input data of requests and output data of responses. In particular, we defined the relationship of the descriptions to the state transition interaction model and defined the semantics of the descriptions with respect to the HTTP methods applied in an interaction. We have also shown how BGPs can be used to define interaction templates, which can be used to formulate search requests for identifying resources.

Research question 3.2 addresses the use of graph patterns for the search of resources. We described how templates as search requests can be matched with descriptions by identifying the sub-graph containment of the involved BGPs. Additionally, we defined several metrics to describe to what degree a given interaction template in a search request matches a resources description. In particular, we defined two metrics based on the used vocabulary terms in template and description, which can be used as preliminary heuristics for matching and ranking. Further, we provided a metric based on the actual containment of triple patterns in the BGPs of description and template, which allows to precisely determine, if a resource matches a given search request and to identify differences.

Research question 3.3 is concerned with the identification of search results in a scalable fashion given a graph pattern-based search request. Thus, we described an architecture to match and rank BGP-based resource descriptions with interaction templates. Specifically, the architecture also updates existing search requests, when new resource descriptions are committed to the system. We have shown how service search is divided into sub-tasks, which can be leveraged in a distributed computing architecture. Based on an algorithm to quickly prune non-matching triple patterns, the proposed architecture achieves scalability with respect to the amount of descriptions and search requests.

In summary we confirm Hypothesis 3. In the context of input and output descriptions we have shown the applicability of graph patterns to describe Linked data resources, which are represented with graph-structured data. While we did not focus on other aspects of services, which could be described, our experiments provide evidence that a scalable system built upon graph pattern descriptions can be designed to search for resources with respect to their technical usability.

6.2 FUTURE WORK

Future work building upon the proposed rule based language to combine data processing and network lookups in conjunction with the parallel execution model (Hypothesis 1) relates to an increased control over the data retrieval and processing. Provenance tracking for the retrieved data items could allow for more expressive restrictions on which resources to retrieve, e.g., a limitation of lookups only to resources that are reachable with a specified number of hops from the initial resources. Thus, the processed data could be even further constrained to the strictly necessary resources that are required to achieve the goal of an application. Provenance tracking for a system that combines reasoning with query answering over interlinked resources presents a challenge, as some statements will be derived from other triples that come from different resources. Therefore, derived triples will have a provenance relation to multiple resources.

A more extensive control over the evaluation of rule-based programs can also relate to a more deliberate scheduling of processing and lookup tasks. The size of the input queues could additionally be taken into account to determine when lookups are executed, to limit the maximal size of the queues and in turn reduce the memory footprint of the program evaluation. Furthermore, the parsing of retrieved data could be separated into an additional thread pool of CPU-bound tasks, thus allowing for more possibilities to assign and distribute available computing power.

Another aspect of future research work can be a higher level of parallelisation: Beyond the parallel evaluation of programs over multiple cores, the applicability of the proposed execution model in machine clusters can be studied. As the communication between machines in a cluster introduces additional overhead in processing, a significantly increased computing power can be applied. Thus the arising challenge becomes the decision making how to distribute tasks over the available machines. E.g., the operator plan could be divided in order to minimise inter-machine communication, or dedicated machines for network lookups could be introduced.

Future work building upon the proposed rule language for the processing, interaction, and manipulation of Web resources according to REST principles (Hypothesis 2) can be concerned with the expressiveness of the rules. Disjunctive graph patterns in rules bodies as well as the possibility to include arithmetic expressions would result in a more expressive semantic. However, an increase of semantic expressiveness comes at the price of increased computational costs and can generally result in undesired characteristics that have to be mitigated. E.g., arithmetic expressions in a rule based program might lead to the creation of new identifiers, and thus inherently prevent the existence of a fixpoint for the program evaluation. The identification of the right balance between complexity and expressiveness to cover a large set of potential use-cases while preserving acceptable runtimes is an important challenge.

Finally, the combination of process modeling approaches with the presented declarative rule-based approach can allow to incorporate explicit choreographies in the data-driven interaction with Web resources. Specifically, the inclusion of suitable choreography representations in the initial graph of an LD-Fu program

in conjunction with the repeated execution of the program to iteratively execute a defined process can be explored. Establishing a bridge between explicit process definitions and declarative rule programs might result in a better control over the execution, while retaining dynamic reactions and fostering parallel execution.

With respect to resource search by leveraging graph pattern-based descriptions (Hypothesis 3) future work also relates to the expressivity of the descriptions and search requests. Independently from the expressivity of the actual resource state representations, more expressive descriptions can allow to encode additional information about the resources. Templates can profit from more expressive descriptions analogously by allowing to formulate more precise search requests. An increase in expressiveness of descriptions and templates requires an extended definition for the matching and ranking. Thus, an approach with more expressive descriptions and templates results in a more complex search process, which in turn can affect the required time to calculate matches.

Another aspect of future work relates to the properties that are described and ultimately used to search for resources. Beyond a description of input and output payload when interacting with resources other non-technical aspects can be relevant for the search of resources, like the costs for using the resources or potential service level agreements offered by a provider. Comprehensive search approaches can integrate such information regarding business and operational aspects with the technical information about the interaction with the resources.

Finally, the provision for inference and reasoning capabilities in the calculation of the metrics can also be the subject of future work. The containment of a pattern in another is not necessarily directly given, but could be inferred by leveraging schema information of the used vocabularies in the descriptions. An extended search approach could retrieve relevant schemata and logically derive matching descriptions for given templates.

Overall there are several research topics that can benefit from the contributions described in this thesis, which have the potential to enhance the interaction and manipulation of Web resources. Given the increasing importance of Web-based applications, advances in this field can provide valuable insights for the use of remote knowledge on the Web.

REFERENCES

[1] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active XML. In *Proceedings of the Symposium on Principles of Database Systems*, Paris, France, 2004. ACM International Conference Proceeding Series.

[2] Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *Proceedings of the International Conference on Extending Database Technology*, Berlin, Germany, 2012. ACM International Conference Proceeding Series.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, Atlantic City, New Jersey, 1967. ACM International Conference Proceeding Series.

[4] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web Journal*, 3(4):397–407, 2012.

[5] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web Journal*, 3(4):397–407, 2012.

[6] Dominic Battre, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the ACM Symposium on Cloud Computing*, Indianapolis, USA, 2010. ACM International Conference Proceeding Series.

[7] David Beckett and Tim Berners-Lee. Turtle - terse RDF triple language. W3C team submission, W3C, 2011. URL http://www.w3.org/TeamSubmission/turtle/. accessed April 14, 2015.

[8] Boualem Benatallah, Mohand-Said Hacid, Alain Leger, Christophe Rey, and Farouk. On automating Web services discovery. *International Journal on Very Large Databases*, 14(1):84–96, 2005.

[9] Tim Berners-Lee. Linked Data. Technical report, W3C, 2006. URL http://www.w3.org/DesignIssues/LinkedData. accessed April 14, 2015.

[10] Tim Berners-Lee. Read-write Linked Data. Technical report, W3C, 2009. URL http://www.w3.org/DesignIssues/ReadWriteLinkedData.html. accessed April 10, 2015.

[11] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. W3C team submission, W3C, 2011. URL http://www.w3.org/TeamSubmission/n3/.

[12] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[13] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - the story so far. *International Journal of Semantic Web and Information Systems*, 5(3): 1–22, 2009.

[14] Daniele Bonetta, Achille Peternier, Cesare Pautasso, and Walter Binder. S: A scripting language for high-performance RESTful Web services. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, 2012. ACM International Conference Proceeding Series.

[15] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.1 (second edition). W3C recommendation, W3C, 2006. URL http://www.w3.org/TR/xml11/. accessed April 14, 2015.

[16] Carmen Brenner, Anna Fensel, Dieter Fensel, Andreea Gagiu, Iker Larizgoitia, Birgit Leiter, Ioannis Stavrakantonakis, and Andreas Thalhammer. How to domesticate the multi-channel communication monster. Technical report, STI2 Online Communication Working Group, 2012. URL http://oc.sti2.at/sites/default/files/oc_short_handouts.pdf. accessed April 10, 2015.

[17] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF Schema. W3C recommendation, W3C, 2004. URL http://www.w3.org/TR/rdf-schema/. accessed April 14, 2015.

[18] Marco Cadoli, Luigi Palopoli, and Maurizio Lenzerini. Datalog and description logics: Expressive power. In *Proceedings of Joint Conference on Declarative Programming*, Gardi, Italy, 1997. Springer.

[19] Jorge Cardoso and Hansjörg Fromm. Electronic services. In Jorge Cardoso, Hansjörg Fromm, Stefan Nickel, Gerhard Satzger, Rudi Studer, and Christof Weinhardt, editors, *Fundamentals of Service Systems*. Springer, 2015.

[20] Jorge Cardoso and Amit Sheth. Semantic e-workflow composition. *Journal of Intelligent Information Systems*, 21(3):191–225, 2003.

[21] Jorge Cardoso and Amit P. Sheth. *Semantic Web Services, Processes and Applications*. Springer, 2006.

[22] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the International Conference on Very Large Data Bases*, Berlin, Germany, 2003. Springer.

[23] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. W3C recommendation, W3C, 2001. URL http://www.w3.org/TR/wsdl. accessed April 14, 2015.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] Joel Farrell and Holger Lausen. Semantic annotations for WSDL and XML schema. W3C recommendation, W3C, 2007. URL http://www.w3.org/TR/sawsdl/. accessed April 14, 2015.

[26] Dieter Fensel. Triple-space computing: Semantic Web services based on persistent publication of information. In *Proceedings of the IFIP International Conference on Intelligence in Communication Systems*, Bangkok, Thailand, 2004. Springer.

[27] Dieter Fensel, Holger Lausen, Axel Polleres, Jos de Bruijn, Michael Stollberg, Dumitru Roman, and John Domingue. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer, 2006.

[28] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, USA, 2003.

[29] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California and Irvine, 2000.

[30] Valeria Fionda, Claudio Gutierrez, and Giuseppe Pirró. Semantic navigation on the Web of data: Specification of routes, web fragments and actions. In *Proceedings of International Conference on World Wide Web*, Lyon, France, 2012. ACM International Conference Proceeding Series.

[31] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[32] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Records*, 34(2):27–33, 2005.

[33] Hansjörg Fromm and Jorge Cardoso. Foundations. In Jorge Cardoso, editor, *Fundamentals of Service Systems*. Springer, 2015.

[34] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI Fall Symposium Series*, 6:45–53, 2007.

[35] Fabien Gandon and Guus Schreiber. RDF 1.1 XML syntax. W3C recommendation, W3C, 2014. URL http://www.w3.org/TR/rdf-syntax-grammar/. accessed April 14, 2015.

[36] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 233(4):120–123, 1970.

[37] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[38] Birte Glimm and Markus Krötzsch. SPARQL beyond subgraph matching. In *Proceedings of the International Semantic Web Conference*. Springer, 2010.

[39] Birte Glimm, Aidan Hogan, Markus Krötzsch, and Axel Polleres. OWL: Yet to arrive on the Web of data? In *Proceedings of the WWW Workshop on Linked Data on the Web*, Lyon, France, 2012. CEUR-WS.

[40] Javier Gonzalez-Castillo, David Trastour, and Claudio Bartolini. Description logics for matchmaking of services. In *Proceedings of the KI Workshop on Applications of Description Logics*, Vienna, Austria, 2001. CEUR-WS.

[41] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[42] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: Combining logic programs with description logic. In *Proceedings of the International Conference on World Wide Web*, Budapest, Hungary, 2003. ACM International Conference Proceeding Series.

[43] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP version 1.2: Messaging framework. W3C recommendation, W3C, 2007. URL http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. accessed April 14, 2015.

[44] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: a benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[45] Andreas Harth and Sebastian Speiser. On completeness classes for query evaluation on Linked Data. In *Proceedings of the National Conference on Artificial Intelligence*, Totonto, Canada, 2012. AAAI.

[46] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over Linked Data. In *Proceedings of the International Conference on World Wide Web*, Raleigh, North Carolina, USA, 2010. ACM International Conference Proceeding Series.

[47] Andreas Harth, Craig Knoblock, Steffen Stadtmüller, Rudi Studer, and Pedro Szekely. On-the-fly integration of static and dynamic sources. In *Proceedings of the ISWC Workshop on Consuming Linked Data*, Sydney, Australia, 2013. CEUR-WS.

[48] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL queries over the Web of Linked Data. In *Proceedings of the International Semantic Web Conference*, Washington, USA, 2009. Springer.

[49] Norman Heino and Jeff Z. Pan. RDFS reasoning on massively parallel hardware. In *Proceedings of the International Semantic Web Conference*, Boston, USA, 2012. Springer.

[50] Antonio Garrote Hernandez and Maria N. Moreno Garcia. A formal definition of RESTful Semantic Web services. In *Proceedings of the WWW Workshop on RESTful Design*, Raleigh, USA, 2010. ACM International Conference Proceeding Series.

[51] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., 2003.

[52] Jan Philipp Hofste. Linked API wrapping with Karma to integrate structured Web services into the Semantic Web. Bachelor thesis, Karlsruhe Institute of Technology, Karlsruhe, 2014.

[53] Aidan Hogan, Andreas Harth, and Axel Polleres. SAOR: Authoritative reasoning for the Web. In *Proceedings of the Asian Semantic Web Conference*, Bangkok, Thailand, 2008. Springer.

[54] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the European Conference on Computer Systems*, Lisbon, Portugal, 2007. ACM International Conference Proceeding Series.

[55] Martin Junghans, Sudhir Agarwal, and Rudi Studer. Towards practical Semantic Web service discovery. In *Proceedings of the Extended Semantic Web Conference*, Heraklion, Greece, 2010. Springer.

[56] Michael Kay. XSL transformations version 2.0. W3C recommendation, W3C, 2007. URL `http://www.w3.org/TR/xslt20/`. accessed April 14, 2015.

[57] Uwe Keller, Ruben Lara, Holger Lausen, Axel Polleres, and Dieter Fensel. Automatic location of services. In *In Proceedings of the European Semantic Web Conference*, Heraklion, Greece, 2005. Springer.

[58] Felix Leif Keppmann and Steffen Stadtmüller. Semantic RESTful APIs for dynamic data sources. In *Proceedings of the ESWC 2014 Workshop on Services and Applications over Linked APIs and Data*, Heraklion, Greece, 2014. CEUR-WS.

[59] Jacek Kopecky, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: Semantic annotations for WSDL and XML schema. *IEEE Internet Computing*, 11(6):60–67, 2007.

[60] Jacek Kopecky, Tomas Vitvar, and Dieter Fensel. MicroWSMO: Semantic description of RESTful services. Technical report, WSMO Working Group, 2008.

[61] Reto Krummenacher, Barry Norton, and Adrian Marte. Towards Linked Open Services. In *Future Internet Symposium*, Berlin, Germany, 2010. Springer.

[62] Günter Ladwig and Thanh Tran. Linked Data query processing strategies. In *Proceedings of the International Semantic Web Conference*, Shanghai, China, 2010. Springer.

[63] Ruben Lara, Miguel Corella, and Pablo Castells. A flexible model for locating services on the Web. *International Journal of Electronic Commerce*, 12 (2):11–40, 2007.

[64] Jonathan Lathem, Karthik Gomadam, and Amit P. Sheth. SA-REST and (S)mashups : Adding semantics to RESTful services. In *Proceedings of the International Conference on Semantic Computing*, Irvine, USA, 2007. IEEE Internet Computing.

[65] Edward A. Lee and Pravin Varaiya. *Structure and Interpretation of Signals and Systems*. Addison-Wesley, 2011.

[66] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views. In *Proceedings of the ACM SIGMOD Symposium on Principles of Database Systems*, San Jose, USA, 1995. ACM International Conference Proceeding Series.

[67] Lei Li and Ian Horrocks. A software framework for matchmaking based on Semantic Web technology. In *Proceedings of the International Conference on World Wide Web*, New York, USA, 2003. ACM International Conference Proceeding Series.

[68] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *Proceedings of the European Conference on Web Services*, Lugano, Switzerland, 2010. IEEE Computer Society.

[69] Frank Manola and Eric Miller. RDF primer. W3C recommendation, W3C, 2004. URL `http://www.w3.org/TR/rdf-primer/`. accessed April 14, 2015.

[70] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the Web: Reasoning over dynamic data. *Journal of Web Semantics*, 25(0):24–44, 2014.

[71] Deborah L. McGuinness and Frank van Harmelen. OWL web ontology language. W3C recommendation, W3C, 2004. URL `http://www.w3.org/TR/owl-features/`. accessed April 14, 2015.

[72] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[73] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In *Proceedings of the International Conference on Data Engineering*, San Jose, USA, 2002. IEEE Computer Society.

[74] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the Fall Joint Computer Conference*, New York, USA, 1968. ACM International Conference Proceeding Series.

[75] Robin Milner. *Communicating and Mobile Systems: Pi-calculus*. Cambridge University Press, 1999.

[76] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Québec, Canada, 2014. AAAI.

[77] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993.

[78] Talal H. Noor, Quan Z. Sheng, Abdullah Alfazi, Anne H.H. Ngu, and Jeriel Law. CSCE: A crawler engine for cloud services discovery on the World Wide Web. In *Proceedings of the International Conference on Web Services (ICWS)*, Santa Clara Marriott, USA, 2013. IEEE Computer Society.

[79] Barry Norton, Mick Kerrigan, and Adrian Marte. On the use of transformation and Linked Data principles in a generic repository for Semantic Web services. In *Proceedings of the ESWC Workshop on Ontology Repositories and Editors for the Semantic Web*, Heraklion, Greece, 2010. CEUR-WS.

[80] Peter F. Patel-Schneider. Reasoning in RDFS is inherently serial, at least in the worst case. In *Proceedings of the International Semantic Web Conference*, Boston, USA, 2012. Springer.

[81] Cesare Pautasso. RESTful Web service composition with BPEL for REST. *Journal of Data and Knowledge Engineering*, 68(9):851–866, 2009.

[82] Cesare Pautasso and Erik Wilde. Why is the Web loosely coupled?: A multi-faceted metric for service design. In *Proceedings of the International Conference on World Wide Web*, Madrid, Spain, 2009. ACM International Conference Proceeding Series.

[83] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful web services vs. "big"' Web services: Making the right architectural decision. In *Proceedings of the International Conference on World Wide Web*, New York, USA, 2008. ACM International Conference Proceeding Series.

[84] Carlos Pedrinaci, John Domingue, and Reto Krummenacher. Services and the Web of data: An unexploited symbiosis. In *Proceedings of the AAAI Spring Symposium Workshop on Linked Data meets Artificial Intelligence*, Palo Alto, USA, 2010. AAAI press.

[85] Carlos Pedrinaci, Dong Liu, Maria Maleshkova, David Lambert, Jacek Kopecky, and John Domingue. iServe: a Linked Services publishing platform. In *Proceedings of the ESWC Workshop on Ontology Repositories and Editors for the Semantic Web*, Heraklion, Greece, 2010. CEUR-WS.

[86] Carlos Pedrinaci, Jorge Cardoso, and Torsten Leidig. Linked USDL: A vocabulary for web-scale service trading. In Valentina Presutti, Claudia d'Amato, Fabien Gandon, Mathieu d'Aquin, Steffen Staab, and Anna Tordai, editors, *The Semantic Web: Trends and Challenges*. Springer, 2014.

[87] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[88] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34:16:1–16:45, 2009.

[89] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.

[90] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the International Semantic Web Conference*, Bonn, Germany, 2011. Springer.

[91] King Jonathan Davis Randal. An overview of production systems. In Elcock and Michie, editors, *Machine Representations of Knowledge*. John Wiley, 1977.

[92] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the International Conference on Multi-agent Systems*, San Francisco, California, 1995. MIT Press.

[93] Jinghai Rao and Xiaomeng Su. A survey of automated Web service composition methods. In Jorge Cardoso and Amit Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2005.

[94] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 2007.

[95] Marco Luca Sbodio and Claude Moulin. SPARQL as an expression language for OWL-S. In *Proceedings of the ESWC Workshop on OWL-S: Experiences and Directions*, Innsbruck, Austria, 2007.

[96] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the Linked Data best practices in different topical domains. In *Proceedings of the International Semantic Web Conference*, Riva del Garda, Italy, 2014. Springer.

[97] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *Proceedings of the International Conference on Database Theory*. Springer, 2010.

[98] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948.

[99] Elena Simperl, Reto Krummenacher, and Lyndon Nixon. A coordination model for triplespace computing. In *Proceedings of the International Conference on Coordination Models and Languages*, Paphos, Greece, 2007. Springer.

[100] Michael Sintek and Stefan Decker. TRIPLE - a query, inference, and transformation language for the Semantic Web. In *Proceedings of the International Semantic Web Conference*, Sardinia, Italy, 2002. Springer.

[101] Steve Speicher, John Arwe, and Ashok Malhotra. Linked Data platform 1.0. W3C recommendation, W3C, 2015. URL `http://www.w3.org/TR/2015/REC-ldp-20150226/`. accessed April 14, 2015.

[102] Sebastian Speiser and Andreas Harth. Taking the LIDS off data silos. In *Proceedings of the International Conference on Semantic Systems*, Graz, Austria, 2010. ACM International Conference Proceeding Series.

[103] Sebastian Speiser and Andreas Harth. Integrating Linked Data and services with Linked Data services. In *Proceedings of the Extended Semantic Web Conference*, Heraklion, Greece, 2011. Springer.

[104] Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. Springer, 2009.

[105] Steffen Stadtmüller and Andreas Harth. Towards data-driven programming for RESTful linked data. In *Proceedings of the ISWC Workshop on Programming the Semantic Web*, Boston, USA, 2012. CEUR-WS.

[106] Steffen Stadtmüller and Barry Norton. Scalable discovery of Linked APIs. *International Journal of Metadata and Semantics and Ontologies*, 8(2):95–105, 2013.

[107] Steffen Stadtmüller, Sebastian Speiser, and Andreas Harth. Future challenges for Linked APIs. In *Proceedings of the ESWC Workshop on Services and Applications over Linked APIs and Data*, Montpellier, France, 2013. CEUR-WS.

[108] Steffen Stadtmüller, Sebastian Speiser, Andreas Harth, and Rudi Studer. Data-Fu: A language and an interpreter for interaction with read/write Linked Data. In *Proceedings of the International Conference on World Wide Web*, Rio de Janeiro, Brazil, 2013. ACM International Conference Proceeding Series.

[109] Steffen Stadtmüller, Sebastian Speiser, Martin Junghans, and Andreas Harth. Comparing major Web service paradigms. In *Proceedings of the ESWC Workshop on Services and Applications over Linked APIs and Data*, Montpellier, France, 2013. CEUR-WS.

[110] Steffen Stadtmüller, Jorge Cardoso, and Martin Junghans. Service semantics. In Jorge Cardoso, Hansjörg Fromm, Stefan Nickel, Gerhard Satzger, Rudi Studer, and Christof Weinhardt, editors, *Fundamentals of Service Systems*. Springer, 2015.

[111] Michael Stollberg, Martin Hepp, and Jörg Hoffmann. A caching mechanism for Semantic Web service discovery. In *Proceedings of the International Semantic Web Conference*, Busan, Korea, 2007. Springer.

[112] Michael Stollberg, Uwe Keller, Holger Lausen, and Stijn Heymans. Two-phase Web service discovery based on rich functional descriptions. In *Proceedings of the European Semantic Web Conference*, Innsbruck, Austria, 2007. Springer.

[113] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data Knowledge Engineering Journal*, 25 (1-2):161–197, 1998.

[114] Rudi Studer, Stephan Grimm, and Andreas Abecker. *Semantic Web Services: Concepts and Technologies and Applications*. Springer, 2007.

[115] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, and Naveen Srinivasan. Automated discovery and interaction and composition of Semantic Web services. *Journal of Web Semantics*, 1(1):27–46, 2003.

[116] Mohsen Taheriyan, Craig Knoblock, Pedro Szekely, , and Jose Luis Ambite. Rapidly integrating services into the Linked Data cloud. In *Proceedings of the International Semantic Web Conference*, Boston, USA, 2012. Springer.

[117] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-based query optimisation for SPARQL. In *Proceedings of the International Conference on Extending Database Technology*, Berlin, Germany, 2012. ACM International Conference Proceeding Series.

[118] Jürgen Umbrich, Aidan Hogan, Axel Polleres, and Stefan Decker. Improving the recall of live Linked Data querying through reasoning. In *Proceedings of the International Conference on Web Reasoning and Rule Systems*, Vienna, Austria, 2012. Springer.

[119] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal. OWL reasoning with WebPIE: Calculating the closure of 100 billion triples. In *Proceedings of the Extended Semantic Web Conference*. Springer, 2010.

[120] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Rik Van de Walle, and Joaquim Gabarr Valls. Efficient runtime service discovery and consumption with hyperlinked RESTdesc. In *Proceedings of the International Conference on Next Generation Web Services Practices*, Salamanca, Spain, 2011.

[121] Ruben Verborgh, Andreas Harth, Maria Maleshkova, Steffen Stadtmüller, Thomas Steiner, and Mohsen Taheriyan. Survey on semantic description of REST APIs. In Cesare Pautasso, Erik Wilde, and Rosa Alarcón, editors, *REST: Advanced Research Topics and Practical Applications*. Springer, 2014.

[122] Tomas Vitvar, Jacek Kopecky, Maciej, and Dieter Fensel. WSMO-Lite: Lightweight semantic descriptions for services on the Web. In *Proceedings on the European Conference on Web Services*, Halle, Germany, 2007. IEEE Computer Society Press.

[123] W3C SPARQL Working Group. SPARQL 1.1 overview. W3C recommendation, W3C, 2013. URL `http://www.w3.org/TR/sparql11-overview/`. accessed April 14, 2015.

[124] Jim Webber. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.

[125] Michael Weiss and G. R. Gangadharan. Modeling the mashup ecosystem: Structure and growth. *R&D Management Journal*, 40(1):40–49, 2009.

[126] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[127] Erik Wilde. REST and RDF granularity, 2009. Available at `http://dret.typepad.com/dretblog/2009/05/rest-and-rdf-granularity.html` and accessed 6th March 2015.

[128] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, Miami, USA, 1991. ACM International Conference Proceeding Series.

[129] Jiehan Zhou, Juha-Pekka Koivisto, and Eila Niemea. A survey on Semantic Web services and a case study. In *Proceedings of the International Conference on CSCW in Design*, 2006.

LIST OF FIGURES

Figure 1 Example of an RDF graph (graphical representation). . . . 16
Figure 2 Linked Data information resource provided by the New York Times (HTML representation) with statements about the document at the bottom. 19
Figure 3 Illustration of an external link between two resources as equivalence relation. (Some prefix definitions omitted for brevity.) 21
Figure 4 Architecture of services provided via APIs, implemented in client application for the consumption by end users. . . 26
Figure 5 Proportion of different protocols used for open Web interfaces. 27
Figure 6 Illustration of the data flow between physical operator plan and request components of a system to process linked programs. 47
Figure 7 Logical operator plan for linked program in Listing 6 and query in Example 13. 52
Figure 8 Physical operator plan for linked program in Listing 6 and query in Example 13. 52
Figure 9 Synthetic tree dataset shapes: (a) a path ($d = 3, b = 1$), (b) a star ($d = 1, b = 3$), and (c) a tree ($d = 2, b = 3$). Leaf nodes are circles with dashed lines. 72
Figure 10 Throughput of of parallel processing of LUBM 200 with custom rule set. 75
Figure 11 Throughput of of parallel processing of LUBM 200 with OWL LD rule set. 76
Figure 12 Throughput of of parallel processing of LUBM 200 with custom rule set; number of cores adapted to number of threads (average of 5 runs). 79
Figure 13 Throughput of of parallel processing of LUBM 200 with OWL LD rule set; number of cores adapted to number of threads (average of 5 runs). 80
Figure 14 Runtime to process locally generated synthetic tree dataset with rules for symmetry and transitivity depending on tree with breadth $b = 2$ and increasing depth. 82
Figure 15 Throughput of system retrieving a tree dataset without deduction rules with breadth $b = 6$ and depth $d = 6$ depending on number of request threads; data is retrieved by network requests with about 2ms delay per request. . . 83

Figure 16 Throughput of system retrieving a tree dataset with rules for symmetry and transitivity with breadth $b = 6$ and depth $d = 6$ depending on number of request threads; data is retrieved by network requests with about 200ms delay per request. 84
Figure 17 Runtime to process retrieved synthetic tree dataset without derivation rules depending on tree with breadth $b = 2$ and increasing depth; data is retrieved via network requests. 86
Figure 18 Average frequency of 1 000 repeated runs of a linked program retrieving a tree dataset with breadth $b = 2$ and increasing depth and materialising triples for symmetry and transitivity. 87
Figure 19 UML Activity Diagram illustrating the dissemination of news. 96
Figure 20 Illustration of a wrapper mediating between a client application and a social network API. 106
Figure 21 State transition of a LDSTS, with excerpts of two states. . . 111
Figure 22 Dataflow of ACME's dissemination system 119
Figure 23 Points of time as distinct states, driven by an LD-Fu program. 126
Figure 24 Interactions with a set number resources. 130
Figure 25 Average runtime from ten executions for different evaluation set-ups to retrieve and modify one set of number resources. 133
Figure 26 Interactions with ten sets number resources. 134
Figure 27 Average runtime from ten executions for different evaluation set-ups to retrieve and modify ten sets of number resources. 135
Figure 28 Average execution and planning time from ten executions of a LD-Fu program to interact and manipulate sets of 1000 resources. 136
Figure 29 Examples of the evolution of a cell (center) according to Game of Life laws. 137
Figure 30 Visualisation of the population (black) on a Game of Life board. 139
Figure 31 Runtime to derive and execute evolution changes in a Game of Life implemented with LD-Fu. 141
Figure 32 Search of recommendation resources by matching template with descriptions. 161
Figure 33 Graphical representation of execution time measurements 168
Figure 34 Graphical representation of measured runtime to calculate cr for a template against 1 000 descriptions. 169
Figure 35 Illustration of the operator plan for the OWL LD rule set (part 1). Recursion to input operator omitted for clarity. . 196
Figure 36 Illustration of the operator plan for the OWL LD rule set (part 2). Recursion to input operator omitted for clarity. . 197

LIST OF TABLES

Table 1 URI namespaces and prefixes of the fictitious company ACME used throughout the thesis . . . 7
Table 2 Overview of HTTP methods (excerpt) and their characteristics. . . . 30
Table 3 Result sizes of programs with different complexity retrieving information about a sports stadium. . . . 44
Table 4 Operators used in the logical and physical plans to process data. . . . 50
Table 5 Average standard error from 5 LUBM runs for the throughput between the marked amount of threads for system A. 81
Table 6 Average standard error from 5 LUBM runs for the throughput between the marked amount of threads for system B. . 81
Table 7 URI prefixes used for social networks and micro blog . . . 96
Table 8 Average runtime from ten executions for different evaluation set-ups to retrieve and modify one set of number resources. . . . 133
Table 9 Average runtime from ten executions for different evaluation set-ups to retrieve and modify ten sets of number resources. . . . 135
Table 10 Average execution and planning time from ten executions of a LD-Fu program to interact and manipulate sets of 1000 resources. . . . 136
Table 11 Metrics for matching template with recommendation resource descriptions as shown in Figure 32. . . . 161
Table 12 Overall execution time to calculate matching metrics and combine results for one template against 10 000 descriptions. 166
Table 13 Execution time to calculate matching metrics for one template against 10 000 descriptions. . . . 167
Table 14 Measurements of runtime to calculate cr for a template against 1 000 descriptions. . . . 169
Table 15 Runtime and throughput for LUBM 200 with custom rule set on Setup A. . . . 198
Table 16 Runtime and throughput for LUBM 200 with custom rule set on Setup B. . . . 199
Table 17 Runtime and throughput for LUBM 200 with OWL LD rule set on Setup A. . . . 200
Table 18 Runtime and throughput for LUBM 200 with OWL LD rule set on Setup B. . . . 201
Table 19 Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup A (part 1). . . . 202

Table 20 Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup A (part 2). 203
Table 21 Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup B (part 1). 204
Table 22 Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup B (part 2). 205
Table 23 Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup A (part 1). 206
Table 24 Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup A (part 2). 207
Table 25 Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup B (part 1). 208
Table 26 Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup B (part 2). 209
Table 27 Runtime and throughput for retrieval of synthetic tree dataset with breadth $b = 6$ and depth $d = 6$ (i.e., 55 986 triple/requests) with about 2 ms delay. 210
Table 28 Runtime and throughput for retrieval of synthetic tree dataset with breadth $b = 6$ and depth $d = 6$ (i.e., 55 986 triple/requests) with about 200 ms delay. 210
Table 29 Runtime and throughput for evaluation of deduction rules for symmetry and transitivity of locally generated synthetic tree dataset (32 TripleWorker and 64 RequestWorker). . 211
Table 30 Frequency and throughput in 1 000 repeated runs for retrieval of synthetic tree dataset and evaluation of deduction rules for symmetry and transitivity (32 TripleWorker and 64 RequestWorker). 211
Table 31 Runtime and throughput for retrieval with about 2 ms and 200 ms delay of synthetic tree dataset and evaluation of deduction rules for symmetry and transitivity (32 TripleWorker and 64 RequestWorker). 212
Table 32 Population, necessary manipulations and elapsed time in a Game of Life of size 10 x 10 over 100 generations. 213
Table 33 Population, necessary manipulations and elapsed time in a Game of Life of size 33 x 33 over 100 generations. 214
Table 34 Population, necessary manipulations and elapsed time in a Game of Life of size 100 x 100 over 100 generations. . . . 215

LIST OF ALGORITHMS

1 Parallel Symmetric Hash Join. 54
2 Control of parallel workers threads 62
3 Adaptive employment of RequestWorker threads 66

4 Determine pattern containment ratio 157

ACRONYMS

AI Artificial Intelligence

API Application Programming Interfaces

BDI Believe Desire Intention

BGP Basic Graph Pattern

conneg content negotiation

foaf friend-of-a-friend vocabulary

HTTP Hypertext Transfer Protocol

HATEOAS Hypermedia as the engine of application state

LD-Fu Linked Data-Fu

LDP Linked Data Platform

LD Linked Data

LUBM Lehigh University Benchmark

N3 Notation 3

OWL Web Ontology Language

QPI Quick Path Interconnect

RDF Resource Description Format

RDFS RDF Schema

REST Representational State Transfer

RPC remote procedure call

LDSTS Linked Data State Transition System

MSM Minimal Service Model

SAWSDL Semantic annotations for WSDL

SA-REST Semantic annotations for REST

SDK Software Development Kit

SOAP Simple Object Access Protocol

SPARQL SPARQL protocol and RDF query language

SWRL Semantic Web Rule Language

SWS Semantic Web Services

Turtle Terse RDF Triple Language

UI User Interface

URI Uniform Resource Identifier

Web World Wide Web

WSDL Web Service Description Language

WSML Web Service Modeling Language

WSMO Web Service Modeling Ontology

W3C World Wide Web Consortium

A

APPENDIX: ADDENDUM

A.1 OWL LD OPERATOR PLAN

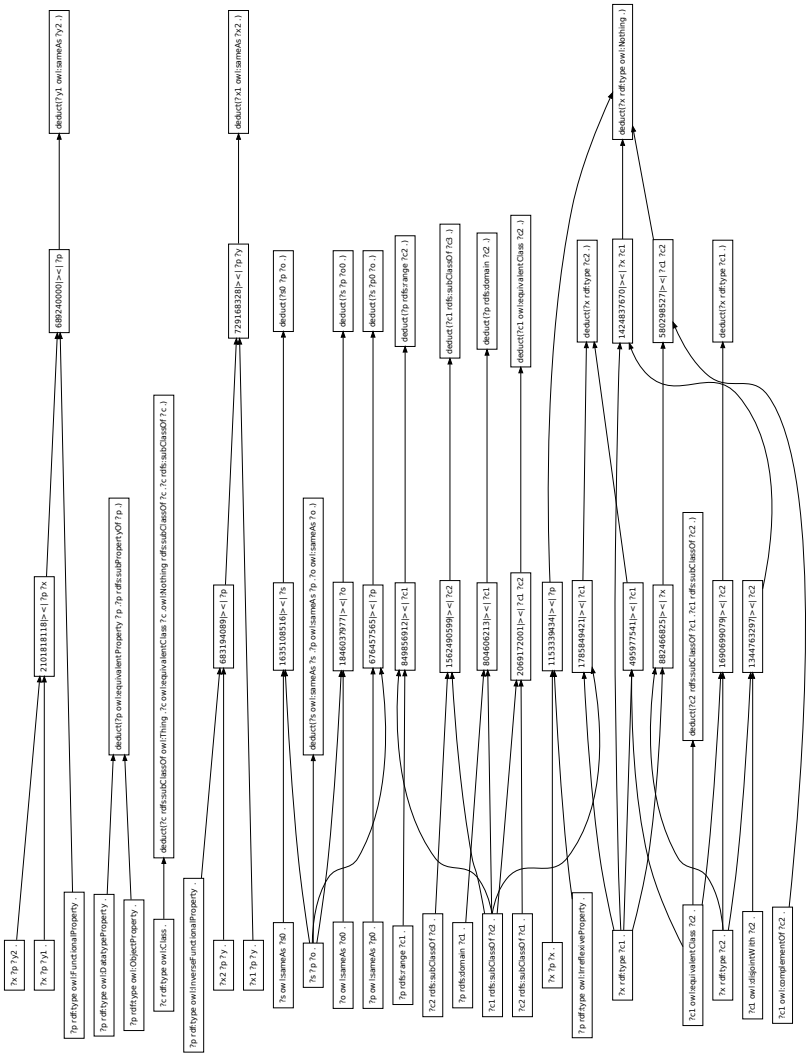


Figure 35: Illustration of the operator plan for the OWL LD rule set (part 1). Recursion to input operator omitted for clarity.

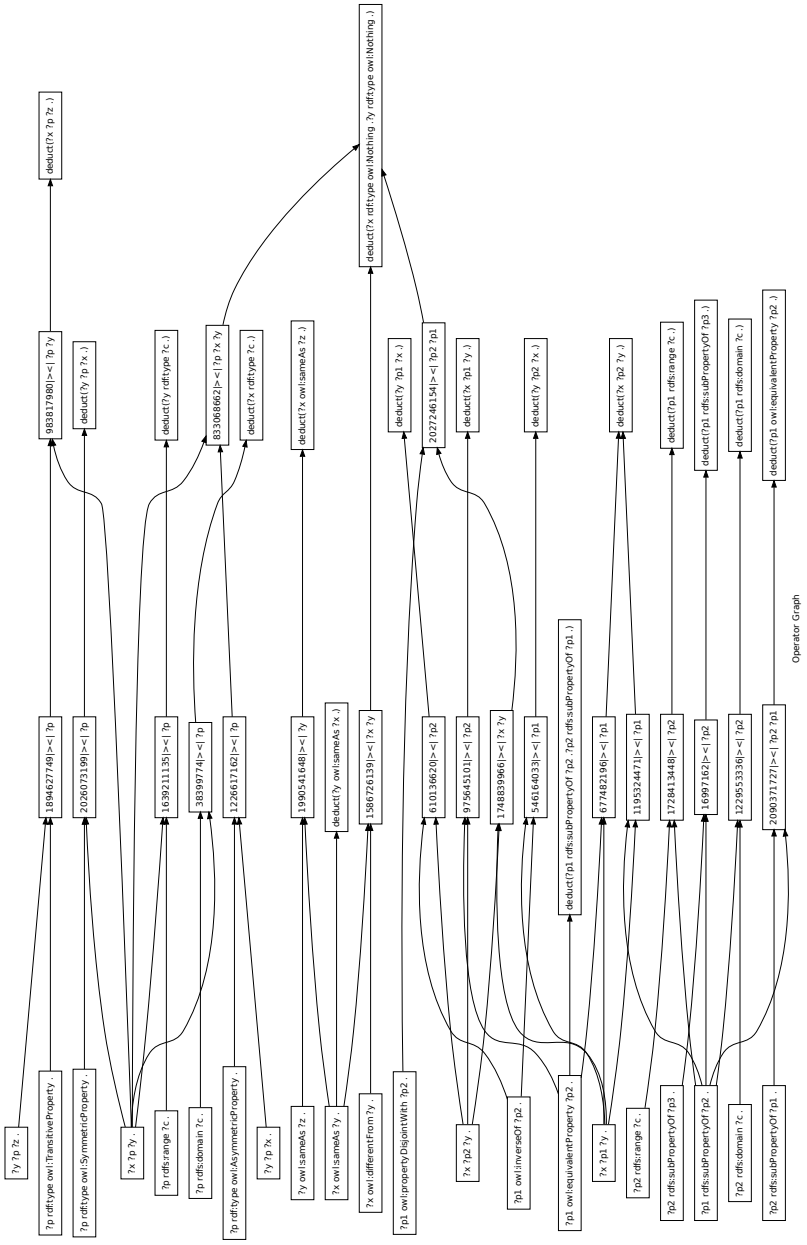


Figure 36: Illustration of the operator plan for the OWL LD rule set (part 2). Recursion to input operator omitted for clarity.

A.2 EXPERIMENT RESULT DETAILS

Table 15: Runtime and throughput for LUBM 200 with custom rule set on Setup A.

	rounds		spinning		blocking single		blocking multi	
threads	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s
1	303562	120074	294785	123649	317829	114684	292085	124792
2	308359	117430	318228	114470	313360	115700	301795	120674
3	194646	182907	213783	161013	207411	173122	205739	171655
4	156697	225411	158859	210364	164593	216989	164977	213987
5	127647	277862	128872	256550	135942	261013	131242	266260
6	108542	321502	109098	305816	113927	309958	109057	313181
7	94371	366653	97821	338620	98073	358228	99407	343476
8	83460	410408	88126	385803	88680	388555	87480	382448
9	77137	449744	79362	402562	81239	430333	74174	452133
10	68451	497971	71490	456719	72261	477196	67070	504792
11	63711	524582	64419	525491	68659	500997	64581	520897
12	61881	557918	65396	514215	62531	551213	61297	535185
13	55164	608850	58254	562525	64088	539922	54234	609598
14	49082	698863	54391	611633	62002	547709	52772	614015
15	49364	686235	51957	638160	52553	645635	48499	689834
16	45758	722627	47399	692989	51033	673924	45728	719438
17	43922	759606	44273	726400	50859	675028	47771	686970
18	43128	784140	43659	752027	49118	692945	44277	704225
19	42257	806191	44677	727360	46618	727693	44584	737333
20	42417	799184	42042	780147	45457	739723	47514	672964
21	41932	818172	42746	766361	47181	729149	40311	801917
22	40367	820325	39660	830718	45485	759206	43242	770049
23	41796	797669	41428	793139	48104	712843	39143	832264
24	38127	877774	39179	821272	42023	808669	36572	882139
25	37347	899954	37565	842284	40346	829426	36494	855318
26	36798	911602	36315	877794	43491	760442	39233	821465
27	40138	787119	35934	887176	38770	872995	33887	952457
28	35230	946569	34588	932942	43275	781909	34273	901910
29	34802	942486	35675	888300	37854	895926	33885	910773
30	35510	933538	33862	965836	39540	814978	33939	914106
31	33087	1007388	34087	919247	37414	901045	32624	972914
32	38400	870413	34924	906424	42392	798802	30173	1023807
33	39545	844389	35963	898735	42870	801760	30835	981300
34	37962	872190	40218	810246	43655	781737	34916	904733
35	37947	883339	37199	862354	46429	739210	34344	955090
36	38216	869642	41383	748652	46367	739592	31224	1014343
37	37966	878298	36562	896321	43896	779878	31266	1020762
38	41090	814230	35762	912001	45939	744704	34078	940783
39	37707	878007	34183	942553	45078	751859	31982	1006692
40	38001	871210	35747	885847	45487	753176	32833	974756
41	37464	894490	35447	895398	43944	778900	29978	1000796
42	40086	832012	34876	892633	42811	791436	32161	1002441

Table 16: Runtime and throughput for LUBM 200 with custom rule set on Setup B.

	rounds		spinning		blocking single		blocking multi	
threads	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s
1	93296	196170	90700	201785	105526	173435	91769	199435
2	96494	189646	92513	197715	102858	177895	96778	188959
3	76337	233688	75556	236549	74194	245061	77016	233391
4	59655	297820	57711	299558	54645	331115	57997	307410
5	46573	374367	48677	342735	46560	382853	46798	375763
6	39897	441285	40367	406795	37657	478434	38891	445711
7	34976	500206	34798	479413	32529	548632	34955	486588
8	31182	551788	32248	516787	27975	642392	30242	564714
9	27921	628471	28227	576131	25601	705654	28767	611075
10	24488	705126	25545	637323	26272	673779	25294	670726
11	23810	736070	23762	701061	23507	758881	23810	726316
12	22784	761581	22985	744125	21815	824400	21367	783442
13	21121	821252	23788	711991	24146	729825	21064	823442
14	20969	828337	21632	781460	22804	776092	20143	864127
15	19416	889921	19676	849745	20669	860776	18040	946012
16	18962	915171	19230	863675	20233	864788	18264	953169
17	17931	956128	19453	873786	20425	854690	17346	970599
18	18534	941294	18449	901640	19750	888860	16178	1067246
19	16771	1027000	18245	903615	19369	892872	15800	1054507
20	16946	1007673	17425	960146	19363	888431	15338	1068579
21	15965	1069462	17233	956962	18397	941593	14969	1106205
22	15973	1054091	16222	1001640	18475	932578	14101	1192394
23	15668	1085089	17280	939637	18015	955027	13923	1193222
24	16169	1050054	15641	1026397	18650	917388	14137	1177309
25	15031	1124005	15559	1018831	17814	948173	13716	1167309
26	16370	1026554	15187	1059894	17666	961958	13195	1249268
27	15270	1083387	15888	1015768	17629	962832	13291	1210689
28	14916	1103401	15356	1024294	19457	884201	14172	1174598
29	14907	1109846	15493	1019493	17268	980803	14359	1151902
30	14690	1116471	14925	1038575	16992	1001960	13879	1207355
31	14320	1152799	15464	1018542	16546	1020310	13383	1246396
32	15106	1103706	18182	855691	18272	930272	13127	1266037
33	14233	1143826	15233	1040497	17502	971762	14463	1127239
34	13736	1201133	14760	1067116	17079	989989	13065	1259645
35	13844	1161407	14904	1046819	17181	977301	12595	1274835
36	13389	1241801	14246	1115501	19004	893877	11706	1418460
37	14167	1159645	15191	1037942	20096	854640	11088	1467688
38	14476	1154628	15349	991350	20125	852135	11655	1381125
39	14931	1117395	15078	1058841	20118	852532	12335	1331713
40	15354	1087166	14592	1079117	20293	840537	12031	1347225
41	14772	1132029	15322	1017706	19709	865862	12946	1269660
42	14704	1133714	15677	993975	20447	838568	11455	1390958

Table 17: Runtime and throughput for LUBM 200 with OWL LD rule set on Setup A.

	rounds		spinning		blocking single		blocking multi	
threads	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s
1	885700	37565	848268	39222	927713	35863	856945	38825
2	646822	51315	634700	51679	657958	50540	643969	51355
3	432254	75419	443419	73098	453832	72304	433937	75164
4	365703	89269	369087	84009	346846	94546	366980	87563
5	303292	106321	313808	98756	307022	106098	300800	106630
6	260532	124132	273185	115016	274438	118126	270116	118209
7	239649	134929	239514	125641	244659	131011	235158	134081
8	219315	147405	218122	142388	220482	146506	208101	153153
9	198690	162141	202238	156421	201175	160150	203068	153208
10	183105	174282	192401	161993	187356	173328	187769	166181
11	173622	184989	175367	176344	178297	180218	174818	174875
12	164923	195587	170206	184782	164696	196436	177525	175821
13	155110	203741	159913	198901	162043	198694	152151	194129
14	156963	204713	151462	202835	147184	216337	155157	194941
15	148064	216273	147634	212875	140705	225520	138974	216847
16	146106	219889	140284	219233	138913	229198	136535	219302
17	133998	237756	134725	229730	136908	235500	137576	215871
18	134077	238675	133102	235868	132879	242044	127607	243682
19	132013	240542	129200	241293	132323	241486	126479	242447
20	126686	253858	123553	252931	126824	255469	119190	256209
21	123849	259557	123690	253148	125635	255901	125884	252447
22	120812	265679	121807	256018	122850	262835	120861	263873
23	127538	252625	120265	261547	120631	267254	118155	268521
24	117740	268656	119045	265427	119645	269249	130057	236543
25	122645	260586	115813	269819	117164	274350	110877	286270
26	116835	271795	111127	278862	113819	281648	111607	273891
27	116168	274834	113989	276101	113201	281762	110644	279482
28	118519	266643	112251	277438	111474	289673	105532	294287
29	113882	280961	108020	289339	110413	292104	107386	286387
30	113247	279866	109017	285402	108512	295247	110283	276395
31	113189	279559	108960	281168	105727	300089	107245	276254
32	108413	286017	105372	290867	107859	298554	102669	285639
33	120513	264029	117862	263658	110345	288165	117523	259293
34	119145	263603	118244	265357	117918	269308	124735	253943
35	124883	257553	117525	264502	116964	272822	113174	272850
36	121715	263531	119739	265937	116892	271981	119902	259673
37	130839	243746	119325	259598	119984	265388	114776	256996
38	120203	266496	120080	261934	121239	262743	115347	269705
39	131675	244746	119654	259212	121166	262065	112284	271689
40	127892	251362	119499	265493	118481	268380	111792	278537
41	132933	238706	119279	266033	116979	274758	110777	277902
42	130378	247075	120148	258918	117550	270652	110611	284617

Table 18: Runtime and throughput for LUBM 200 with OWL LD rule set on Setup B.

	rounds		spinning		blocking single		blocking multi	
threads	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s	time (ms)	Triples/s
1	242803	68809	242377	68930	257549	64869	250048	66815
2	212025	78788	205404	81301	198239	84269	190080	87865
3	157688	104568	157118	104600	156210	106149	156339	104591
4	124470	131924	124796	128562	125144	131819	119644	136663
5	105737	153233	105573	147945	104148	157925	105521	154011
6	92039	176515	88531	178350	87541	188500	87294	184070
7	80466	201943	81258	193206	82109	198506	78921	203806
8	74013	218417	74585	205754	70070	232794	70607	220412
9	68839	235024	64846	237697	62355	264485	65990	241741
10	61900	261907	59429	267594	58708	281774	60728	261681
11	59196	275605	59251	265321	58336	277288	55714	284609
12	56044	288503	54775	287235	52485	314224	56478	277193
13	52929	304997	52147	303185	51169	322164	50926	309189
14	51833	312171	51053	312135	49926	324946	50453	315458
15	49658	327140	48371	320774	49385	330991	46241	338621
16	50727	319929	46462	336649	46411	352743	46689	342249
17	46727	347062	45575	347028	45715	356030	46627	346947
18	45748	354170	44159	357654	44638	364486	46663	339926
19	46146	351537	44333	352556	44554	360562	42627	378328
20	45099	357877	41817	375925	44127	366442	41370	383092
21	43523	369607	42620	371625	42987	375238	43956	359219
22	44438	363872	41156	381277	42353	380942	41670	382025
23	43823	365953	41610	381794	42356	378244	39813	399031
24	43760	367720	40298	381034	42467	381934	38205	410314
25	44045	364293	39943	390496	40096	402941	43137	368007
26	41414	383681	40329	388061	39289	408144	38837	398155
27	40536	389696	40216	387296	39882	398639	38201	413071
28	41662	384926	39771	384560	39548	406383	40924	383573
29	40559	391726	38145	405208	38422	417067	41182	385396
30	40179	394075	36811	419888	37666	426681	38303	417085
31	39568	400474	38846	396760	36076	444600	36922	422455
32	38838	407997	36455	423903	38348	416199	37080	417497
33	41514	382324	38080	405748	38529	413044	36937	422605
34	38976	407513	38317	396575	37071	430414	39397	399940
35	38421	410025	36834	422459	36961	432452	38629	399622
36	39844	396217	37648	404151	37826	422340	36760	429634
37	42733	374615	40668	381372	39332	408124	38394	411579
38	41113	385100	40760	377421	41243	389848	40700	385998
39	43521	365646	41467	374087	41602	385179	39780	394210
40	42967	369907	42033	366569	42791	374554	41536	376792
41	43919	364747	41329	379640	40536	396147	38367	411297
42	42795	373525	43482	358844	42058	379808	39736	401216

Table 19: Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup A (part 1).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
1	289784.6	126168.8	3166.6	285209.2	128320	3692.6	376264.4	96988.8	1508.2	281673.8	129681.8	2712.7
2	223796	163394.6	4340.4	218182.2	167131	1918.9	243286.6	150621.6	4970.8	215640	169275.8	3227.4
3	147730.2	246606.6	5210.4	148584.2	245122	4164.3	165035.6	220731.8	3389.9	153881.4	236626	4564
4	114644.8	317086	5408.5	114865.8	316424.8	4894.4	127414.4	285659.4	4573.1	119092.2	305285.8	6685.4
5	94245	385116	5529.9	93641	387885.4	6047.2	104857.2	346796.4	6859.1	97934.8	370362.6	6954.1
6	79738.2	454694.6	8833.4	79218.2	457089.2	7412.5	88666.4	409415.6	6726.1	83826.6	432003.6	8804.6
7	68719	525082.8	7642.6	69570.6	520025.6	8959	76162.8	475651	9275.7	72646	497886.6	13247.8
8	61434.8	586347.2	9478.3	61022.6	591282.6	12186.7	68281.8	530063.8	7325.5	65577.8	550792.4	12289.6
9	68630.4	499781	17873.6	68246.8	504588.8	17783.9	79374.6	443810	12226	75635	440657.2	5513
10	66782.6	507067	13696.6	65842.6	506218	10539.5	74409	465189.2	7150.7	69388	471471.8	6414.5
11	62976.6	541446.4	3284.4	62230.6	527594.8	6581.2	71394	479896.2	5438.1	65640	485641.4	9264.5
12	59230.6	570409.2	7227.8	58470	563841	4676.6	71339.2	485842.6	8198.7	62611	509849.2	3546.8
13	56325	605762.2	6225.2	56819	573791.6	11448.3	67102	512603	6389.2	56642.6	574738.2	10212.3
14	52483.6	650775.2	6647.9	54128.6	609508.6	14503.6	63761.2	538079.4	4737.2	54448.4	608268.4	17405.2
15	50662.4	668243.2	8026.7	52081	631249.8	8532.4	60993.8	565258.6	6154.1	52660.8	624999.6	6093.2
16	47998	704205.4	11376.8	47218	695735.6	9933.4	59990.8	576515.2	5913.6	49654.6	663383.6	9779.9
17	47138.8	718274.6	6791.4	46171	706456	5208.9	57035.6	597074.2	7361.8	47670.8	682922.4	6582.5
18	45398	742047.8	10748.8	45400.4	709571.2	4266.3	53971.8	631690	12907.5	47498.8	682233.8	4520.9
19	44869.6	754927.4	14016.5	46281.6	711077.2	10420.9	52562.8	654925.8	6438.8	44874.8	728637.6	8476.4
20	43242.6	780311.4	8219.6	42313	771394.2	12024.6	51166.6	671799.6	12977	42579.6	764632.6	13475.1
21	42440.2	792567.4	6198.2	43094.8	766477	8878.9	49913.8	682214.8	9048.1	41679.8	781264.6	23097.3

Table 20: Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup A (part 2).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
22	41192.8	820197.2	13153.2	42262.6	782615.8	12906.2	48020	710227.2	10157.9	41508	782046.2	14914.5
23	40168.6	837258.4	11805.2	41628.2	786311.8	12412.5	48556.6	700440.2	10329.4	38717.2	820934	25579.4
24	37572.2	892990.8	12610.7	39286.8	811866.8	10937.8	47625.8	711684.8	11714.6	39139.4	809363.8	25165.2
25	37741.4	887081.8	11215.1	39026.2	831927	14926.5	46962.2	725798	10030.8	38624	828506.6	11201.8
26	36667.8	902537.4	17094.5	38356.6	845358.2	13272.9	44713	758394.8	14387.3	36719.8	850158.4	21448.1
27	37935.8	873287.4	24746.2	37799.4	858283.8	17886.3	43874	774187.4	8702	35773.6	889093.4	19083.2
28	37104	888058.6	9225.9	36546	882994.6	21553.8	42928.2	788441.6	13092.4	35751.2	878676.6	8347.2
29	35470.2	940078.6	2440	36111.2	897187	3768.6	42645.4	796436.8	3642.9	34034.2	905212.2	9868
30	34819	952452.6	20013.6	35204	907666.8	13448.9	42014	806643.8	10320	33735.8	922158.2	24268.9
31	34674.8	947287	15557.7	33820.6	949605.4	15534.6	40885.4	833283.4	11926.1	33725.8	931261.8	21397.4
32	37214	884637.2	19913.6	34158.8	932793	21842.9	40716	832327.2	7732.4	31662.8	991226.8	15084.5
33	37532	882037.4	13931.1	35401.2	892280	12993.9	45897	744039.2	6738.4	33029.8	944692	19959.2
34	38132.4	845418.8	18024.6	34672.2	915218.6	12437.3	45680.2	742877	3770.1	34201.6	941496.6	14879.4
35	38728.4	851905.4	28473.7	34255.8	924082.2	7732.2	46719	726040.4	14435	33932.6	965144	12205.9
36	40296.6	822940.8	7894.1	37887.4	876868.6	9981.1	46150.6	742556.6	14573.7	32183.6	975564.8	15283.2
37	38626	841282.8	12336.5	37088.6	865727.6	21579.4	44535.4	753360.8	9692.7	33892.2	934154.2	6418.1
38	37495.4	838003.8	30162	34745.4	921445	9975.7	45379.2	750963.4	12477.6	34074	929576.2	17138.1
39	41058.4	815325.4	21729.8	35830.4	899001.6	5035.3	45116.4	743966.8	15898.4	34123.6	925237.6	25806.8
40	38097	874284.4	13145.7	35711	895704	10982.2	45911	741431.4	19382.8	34050.4	926984.6	17057
41	39380	845319.8	20186.4	36307.2	890636.4	25848	44540.6	762820.6	12316.2	33422	938769.8	20756.3
42	38526.2	864124.6	16155.4	37162.2	872352	21305.7	45959.2	739163.8	15746.2	32533.4	955767.2	20674

Table 21: Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup B (part 1).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
1	92664.6	197508.2	217.7	90816	201531.6	417.4	116792	156738	1011.3	90977.8	201171.4	309.6
2	78884.2	232029.4	1171.2	79454.6	230353.6	1159.6	92235.4	198434.4	1179.7	77930.8	234859.6	1515.4
3	54198	337430	1058.8	54354.2	336115.2	1214.4	63294.2	288943.6	1845.4	55455.8	329397.4	1359.2
4	42101.4	433659.2	719.5	41788.2	436330.2	1747	48026.8	380458.6	1403.3	42544.6	428801.2	1446.9
5	34309.6	532117.6	2269.4	34328.6	531125	2865.5	39420.2	463271.6	2792.7	34864.2	522144	2701.7
6	29154.8	625450	3797.1	29138.8	624295.6	2328.9	33209	549490	2871.3	29487	615877	2712.6
7	25483.4	714660.6	2607.2	25631.8	708988	4176.6	29281.6	621912.6	2179.8	25740.2	705105	2732.3
8	22633.8	804022.2	3673.6	22688.8	797754.2	4184.1	26077	698250.6	8142.3	22983.8	786723.2	2612
9	20482	887897	5573.6	21109.4	858639	4556.2	23989.6	756657.4	3281.9	20858.2	865538	3231.6
10	22000.8	798384.2	5307.5	22173.8	775124	6166.8	24945.8	716694.8	8723	23664.6	727868	8349.7
11	22900.6	757557.6	5276.4	23325.4	719792.2	4225.3	26498.4	661997.4	7336.2	22734	726501.4	6590.4
12	22311.8	768815.4	8320.4	22112	738679.2	4107.8	26659.4	650323.2	7941	22383.8	737003	3864.5
13	20738.6	817297.4	8418	21533.6	755266.2	5256.1	26074.8	669185	8975.4	20901.8	798828	11557.9
14	20603	824132.4	12274.1	20719.2	782058.6	9805.3	25111.8	692626.6	10099.8	19938.6	826100.4	12654.8
15	19655.4	864856.4	11471	20486.8	797028.2	9847.1	24854.8	700636.8	8324.6	19055.8	863305.8	16995.4
16	18869	900576.2	16216.6	20072.2	805577	5800.7	23613.6	731773.8	5635.3	18601	896528.2	10000.1
17	18221	928372.2	15720.8	18726	861176	8846.4	22727	758216	6906.9	17253.4	960667.8	14914.7
18	17529.4	963980	15489	18660.2	869090.6	19665	22521.8	764852	10431.4	16183.6	1012568.4	8744
19	16721	1010328.6	8248.6	18633.2	876941.2	27750.4	22539.4	763909.8	8927	17124.8	969285.2	15845.8
20	16750.6	1010455	8982.7	17408.6	926137	10505.5	21941.6	788101	8974.6	16968	990337.4	25581.8
21	16873.4	997016.8	16713.7	17255	931103.2	11831.5	21702.4	797310.6	9291.4	16063.2	1038905.2	13564.6

Table 22: Runtime and throughput for LUBM 200 with custom rule set and number of active cores adapted to number of threads on Setup B (part 2).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
22	16633.8	1017858.2	8744.4	17098.8	937052.2	15643.2	21103.4	815035	7861.6	16020	1057814.2	8724.2
23	16327.2	1037518.6	12831.7	17236.6	928745.2	12188.4	21882.8	791426	9661.4	15731.8	1064343.2	35871.4
24	16305.4	1025439.6	22980	16669.6	960803	12529.6	20972.8	817234	9071.7	15267.6	1081124.2	19480.5
25	15190.4	1109247.8	27731.8	16523	964244.2	11302.6	20947.2	821455.4	13606.7	16208	1022127	32106.5
26	15398.2	1080177.2	16623.9	16009.8	992928.8	5068.4	20918.2	817152.6	4213.2	15601.6	1066184.6	33648.8
27	15583	1076187.4	26707.6	15999	997139.2	13388.4	20134.2	855442.6	21454.7	15162	1073943.6	31475.4
28	15255.6	1102109.4	48964.6	15709.2	1006715	17195.6	20515.6	833417.4	6683.3	16004	1023384.2	21350.1
29	15344.2	1071596	12977.5	15709.4	1003361.8	20257.8	19350.6	891249	7555.8	14342	1157547.4	39493.4
30	14578.6	1143521.6	18406.1	15415.4	1018179.6	11361.5	19446.4	878117.6	5691.7	14500	1150770	45779.3
31	15795.4	1044952	21137.1	15398.6	1009849	16482.5	19429.2	874802	10430.9	14478.6	1128359.4	33638.3
32	15239	1072783.4	14758.4	17635	890355.6	28670.2	20657.4	825480.2	7677.1	13697.6	1205258.2	26699.4
33	15027.8	1083162.4	14393.5	15039.6	1046582.8	9676.7	18864.2	904055.8	7810.4	13850.6	1179185.2	27028
34	14469.2	1121800.2	8819.3	14945.2	1038565.8	10613.3	19138.2	885901.4	8829.8	13462	1209563.8	25585.5
35	14327.6	1144455.8	17994	14632.8	1041731	6824.8	18867	901741.6	7266.8	12294.6	1317569.4	66388.9
36	13559.2	1204395.6	9805.6	14733	1066210.6	13922.1	18477.8	919102	7478.1	12153.8	1342245.4	35627.1
37	14582.4	1140574.2	23628.4	14946.2	1049155	10559.7	20305.8	846594.4	15298.6	12392.2	1308357.4	34962.5
38	14779.8	1121268.6	13051.5	15110.6	1041937.4	12189.9	20180.6	849496.8	8733.2	12173.2	1306193.6	26413.2
39	15125	1098978.2	24663	14893.2	1062680.8	21623.2	20209.6	848354.8	7787.5	12460.6	1288277.2	25736.8
40	15223.6	1089028	16784.8	15222	1040125.8	5460.2	20213.8	850137.4	8110.1	11839.8	1361368.8	40859.1
41	15253.8	1091602.2	8216.2	15335.2	1039799.6	23215.3	19959.4	863145	6244.5	12514.4	1297702.8	23097.2
42	15101.8	1104468.8	11064.7	15648.6	1020021.6	3460.6	20679.8	829554.2	14295.2	12224.2	1333699	17763.6

Table 23: Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup A (part 1).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
1	861343.4	38748	976.4	846061.6	39455.8	1049.7	922745.8	36132.6	751.7	849766	39271.8	17562.9
2	532885.8	62554.4	1273.1	526223.6	63284.2	911	546064.8	61038.6	1165.4	556960.6	59969	26819
3	383931.4	86684.4	1694.3	376412.4	88408.4	1751.1	390395.8	85366.6	1903.6	378753	87934.4	39325.5
4	301399.4	110326.4	2475	296446.4	112131	2218.5	308788.4	107842.2	2701.6	298287	111483.8	49857.1
5	248571.8	133605.4	2723.9	249046.2	133370.8	2492.1	260617.8	127634.8	2979.3	252318	131637	58869.9
6	216003.6	153652	3261	214953.6	154600.2	3325	223387.4	148699.2	3115.2	217556.6	152554	68224.2
7	197089.4	168327.6	3023.7	191991.6	172984.8	3900.9	197653.4	167999	3302.5	199728.2	166222.6	74337
8	175878	188327	4735.3	176893.6	187117.6	4208.6	177407.2	186928	3196.9	182295.6	182089.2	81432.8
9	181316.6	178732.8	4355.2	179206.8	179124.6	3472.8	181064.4	179560	4003.1	183669.8	171032.2	76487.9
10	176518.4	180166.6	2795.2	177135.8	178064.6	4188.1	175218.8	183603.6	4356.6	174973.8	178102.4	79649.8
11	170735.6	185997.2	2247.1	172599.6	180184.8	1612.7	172962.8	184955	1900.5	170273	177975.2	79592.9
12	168014	189304.2	1572	167182.2	185510.6	1523.9	168785	190129.4	995.1	168472.2	183388.2	82013.7
13	157321.6	202479.8	2042.1	161711	192396.8	1043.3	159621.8	201118.6	1435.3	152695	198331.6	88696.6
14	149011.6	214370.8	1378.5	153299.6	202131.8	3051.2	149907.4	213245.8	2119.4	152018.4	202685.8	90643.8
15	148904	213228.2	1754.1	150056.6	209960.8	1820.2	145145	221711.4	659.4	142632.6	211712.8	94680.8
16	146186.2	217748.8	2210.4	142192	217657.6	1879.1	146032.6	218038.6	2768.3	137834.8	217330.4	97193.1
17	137307.6	230624.8	3582.4	139351	221575.6	960.1	139853.8	228854.8	1728.7	135911	226095.2	101112.8
18	134085.6	237839	3674.3	134730.6	233224.8	1445.9	135076.4	236798.4	2295.8	131235.2	236458.6	105747.5
19	129536.2	246116.8	3218.8	133699.8	233894.6	3076.9	130985.6	244536	1425.5	124073.8	252440.8	112895
20	127715.8	249818.6	2672	128863	243083.4	2769.9	125141	257073.2	1170	120773.8	258488.4	115599.5
21	124447.8	256858.4	3258.4	123102.8	254420.4	2774.3	119601.4	268008.8	2331.4	120793.8	259733.4	116156.3

Table 24: Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup A (part 2).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
22	122046	261836	1715.1	119738.2	262728.6	1421.8	121496.6	263035.4	2028.2	118599.8	263651	117908.3
23	118344.6	269513.6	2830.4	118434.2	263790.4	872.7	117735.6	271017.4	2065.9	114572.6	270734.8	121076.3
24	119259.4	263825.6	2639.4	118273.2	262089.2	2374.9	119726.8	266045.2	3011.8	114533	267321.2	119549.7
25	117222.6	271196.6	2346.2	113990	272866.4	4600.4	112733.2	282153.6	1205.2	110553.4	272647.6	121931.7
26	118282	267928.6	2771	113919.6	272056	2805.9	114492.6	279013.8	1778.7	111585.4	274120.8	122590.5
27	116268.4	271575	6641.9	112600.4	273572.6	2759	112490.8	284208	2533.5	107948.4	281907.6	126072.9
28	118359.4	267676.4	4259.3	111159	277906.4	3143.9	112219.2	284295.2	2538.3	109518.6	279291.6	124903
29	111973.2	282591.2	1883	112313.2	277019.6	3983.2	110911.2	288358	1996	109743.2	274116.6	122588.7
30	110702	286936	5137.8	108699.4	283316.8	2131.5	110189	290211	1955.5	106030.2	285368	127620.4
31	108649	289786.6	4226.6	107965	285776.8	3149	110760.2	288688.8	3037.5	105084.6	285767.6	127799.2
32	110094	285863	4728.2	106070.8	286005.8	3282.9	107328	296469	583.2	103053.2	283967.6	126994.2
33	118343.6	270310.8	3622.6	116638.8	268109	1328.9	118438.6	269417.4	2098.6	107640.2	285642.2	127743.1
34	118430.4	269431.6	2743.7	118824.8	263272.6	2889.6	118859	270441.2	2847	111132.6	279805.6	125132.9
35	122810.8	259573.2	1941.3	118498	264296.2	2473.9	118917	268536	1298.4	113441.2	270655.4	121040.8
36	121355.6	263730.2	3834.3	118226	266093.8	4639	120278.2	266279.8	2101	113940	269770.6	120645.1
37	126526	253300.6	2763.5	120687.4	259458.6	2349.9	118496	268920	1553.3	114535.8	267629	119687.3
38	120318.8	264923.2	1803.6	117828.4	266428.8	3847.2	118736.2	269314.8	3216.7	117530	261004.2	116724.6
39	123495.6	259582.6	2429.1	119173.2	262692.8	2620.7	119734.6	267510.8	1622.7	113627.4	273303	136651.5
40	123739	258475.2	3191.9	119095.8	264430.8	992.2	117730.8	271778.6	1774.2	113501.2	271275.6	156621
41	121791.2	263107	5920.6	118275.2	265156.6	926.6	119418	268234.8	1752.2	118563.2	260731	184364.7
42	123176.4	260149.8	2008.8	117167.6	268642.8	2623.4	118831.2	270663.8	2767	116938.8	265764.4	265764.4

Table 25: Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup B (part 1).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
1	244196.6	68417.2	104.5	244028.4	68464.6	122.7	262694.8	63599.6	107.8	241905.6	69066	151.3
2	169283.2	98695.8	321.2	169519	98555.8	324.5	176587.2	94602.8	168.2	168028.4	99423.4	409.7
3	121349.6	137605.2	633.5	121007	137916	269.1	129442.4	128982.8	208.4	121841.6	136972.8	877.5
4	96620.2	172702.8	641.1	95840.8	173958	415.6	102844	162292.2	426.3	95945.4	173671.4	657.8
5	82305.6	202610.6	536	79709.8	209002.2	583.3	85821.8	194313	748.7	80847.8	205760.4	1602
6	70783.8	235447.4	488.2	69468.8	239528.2	740.4	73812	225793.4	666.9	70050.4	237531.8	2082.5
7	63444.6	262721.8	673.5	62038.8	268303.4	501.6	65660	253871.8	527.1	63019.2	263795.6	3151.7
8	58885.2	282675	713	56461.4	293856.2	292.4	59475.4	279993.2	630.8	57287.6	289502.4	1880.2
9	54128.8	307374.2	575.5	52243.2	317832.6	971.3	54754.6	303844.8	451.1	52241.4	317640	3407.3
10	55863.4	293452	988.2	54682.2	295530.4	1207.8	55615	296868.4	1016.8	54415.6	296363.2	1847.5
11	57344.8	281154.2	2385.5	56184	280662	2712.2	55834	291921	2170.5	55215	287439.4	4092.7
12	56522.4	281885	3089.2	54776.6	284614.6	2181.3	55543.4	290936.2	1212.5	54337	284825.8	5927
13	54254.2	294722.4	3342.4	53480.2	288596.8	3849.1	55751	289669.4	2369.8	51773.4	294286	3124.5
14	52556.4	304566.6	2364.4	51522.6	300903	2477.4	53965.8	298786.6	2109.8	50196.4	305594.8	4836.1
15	50891.6	313652.4	2111.2	49645	312692.8	1858.8	51847	311940.8	2022.1	48739.8	313328.8	3071.7
16	49892.2	321086	3707.4	47804	323461	1411.8	51206.6	314013.2	3500	45777.4	342991.2	4362.3
17	48851	327075	330.2	47245.6	327846.8	2344.8	48216.2	335490.6	3107.8	44717.8	339266.6	4323.4
18	47426	335193	1802.2	45151.8	341823	1485.7	47640.8	337212.8	2342	44180.6	347126.6	3310.3
19	46919.6	340420	1911.8	44923.4	346727.4	2701	46815.2	344549.4	3047	43090.6	359943.4	5472.1
20	45940.6	350345	2288	43832.2	353886.8	1973.4	45165.4	357950	1175.2	43873.6	358380.4	5314.2
21	45847.8	349695.6	2109.2	44230.4	351921.6	3160.7	45008.6	359132.2	1759.4	43887	358990.8	8205.7

Table 26: Runtime and throughput for LUBM 200 with OWL LD rule set and number of active cores adapted to number of threads on Setup B (part 2).

threads/	rounds (avg of 5 runs)			spinning (avg of 5 runs)			blocking single (avg of 5 runs)			blocking multi (avg of 5 runs)		
cores (up to 32)	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err	time (ms)	Triples/s	std err
22	45725.2	348765.4	2163.3	42846	364323.2	5348	44960	358202.6	3284.4	41146.2	382028.8	5044.1
23	45065.8	355625.8	2307.3	41526.2	375397.8	2688.7	43338.6	371637	2085	41348.2	383111.4	4124.6
24	44344	361364	3241.3	42555.8	362990.8	3323	43227	373365	5027.9	41114.4	385008.4	2995.2
25	43319.8	368536.6	3906.4	41071.4	378793.4	7122.9	41695.6	387228.4	3017	40332.6	391242	11271.9
26	43881.8	362723.2	6452.6	40376	385396.6	3447.3	43146.2	372951	3577.7	39453.2	398338.8	6211.7
27	41413.8	384934	5304.9	40403.4	384258.4	3889.1	41984	383368.6	5471.8	38720.8	405232	8594.6
28	41539.2	381487.2	6248.7	40331	379570	7170.2	41509.4	385382.4	4630.6	38374.6	408342.2	7499.1
29	41410.6	381951.2	4978.2	39597.8	387904.2	5589.5	40834	391232.4	2577.6	40153	388599.8	4371
30	39886.6	400206	2290	38430	399819.8	2854.5	39826.8	403372.8	4050	36929.4	425146.2	4521.7
31	40706.4	387876.2	3853.3	38377.8	400396.6	6167.9	39571.4	405080	3520.9	38587.6	403927.2	4590.4
32	40737	389515.8	4470.8	37666.6	407261.4	4454.2	39915.2	400468.8	2286.5	37947	409740.4	6643
33	40456.2	392232	3998.5	38911.2	394930.6	3969.5	38605.2	414931	2118.4	39298.6	395888.4	9824.1
34	40194.6	395397.2	5935.8	38328.4	400487.8	3692.1	38457	415385.8	1498.4	37906.4	406689.2	2656.2
35	39556.8	402397.2	3767	38289.4	399746	1713.4	37438	427218.4	4315.6	37521.2	418864	3809.9
36	38718.8	408573.4	1764.4	37870	405318.6	2328.2	38039.2	420438.2	4311.4	38195.6	408696.4	8015
37	41451.6	383615.2	3903.6	41155.4	378919	3633.6	40139	399817	2984.2	40435.2	390243.8	4366.5
38	43039	372520.4	5807.6	41426.8	373185.4	3013.4	41032	392094	2983.9	40962.2	385594	6308.2
39	43472	368308	3962.4	42603.4	364671.8	1936.4	41484.6	386385.2	2681.6	40093.2	392755.8	1992.4
40	43795.8	366223	2945.5	41672.8	371471.6	1834.5	41542.8	387242.6	3816.2	41509.8	378162.6	6342.1
41	42369	377196.8	4741.6	42433.4	364495.6	4115.6	40664.2	393763	491	41559.4	380285.6	6420.3
42	43630.2	367395	2045	41882.4	372395	2330.2	41556	386111.8	4273.4	40962.2	383231.8	3220

Table 27: Runtime and throughput for retrieval of synthetic tree dataset with breadth b = 6 and depth d = 6 (i.e., 55 986 triple/requests) with about 2 ms delay.

request	rounds		spinning		blocking single		blocking multi	
threads	ms	Triple/s	ms	Triple/s	time (ms)	Triple/s	ms	Triple/s
32	9777	5726	9413	5947	9438	5931	9436	5933
64	7477	7487	7475	7489	7388	7577	7401	7564
96	7134	7847	6864	8156	6940	8067	6958	8046
128	6838	8187	6428	8709	6432	8704	6514	8594
160	7277	7693	6415	8727	6326	8850	6414	8728
192	6797	8236	6153	9098	6190	9044	6163	9084
224	6946	8060	6082	9205	6018	9303	6098	9181
256	6710	8343	6070	9223	5923	9452	6015	9307
288	6754	8289	5981	9360	5843	9581	5957	9398
320	7600	7366	5877	9526	5817	9624	5822	9616
640	9772	5729	6467	8657	6353	8812	5907	9477
960	10858	5156	5846	9576	5717	9792	5994	9340
1280	13248	4225	5833	9598	5916	9463	6066	9229
1600	13008	4303	5844	9580	5786	9676	5792	9666
1920	16002	3498	5709	9806	5842	9583	5970	9377
2240	16370	3420	5805	9644	6191	9043	6082	9205
2560	19537	2865	5978	9365	8516	6574	6076	9214
2880	22748	2461	5967	9382	6783	5953	6454	8674
3200	25064	2233	6162	9085	11345	4934	6534	8568

Table 28: Runtime and throughput for retrieval of synthetic tree dataset with breadth b = 6 and depth d = 6 (i.e., 55 986 triple/requests) with about 200 ms delay.

request	rounds		spinning		blocking single		blocking multi	
threads	ms	Triple/s	time (ms)	Triple/s	ms	Triple/s	ms	Triple/s
32	353262	158	352699	158	352630	158	366291	152
64	177323	315	176764	316	176727	316	191706	292
96	118586	472	118078	474	118084	474	128829	434
128	89416	626	88820	630	88785	630	100711	555
160	71985	777	71164	786	71271	785	80572	694
192	60164	930	59458	941	59443	941	69513	805
224	51728	1082	51059	1096	51008	1097	59263	944
256	46662	1199	44851	1248	44790	1249	52197	1072
288	40640	1377	39985	1400	39944	1401	48977	1143
320	37308	1500	36226	1545	36164	1548	45452	1231
640	37799	1481	36211	1546	36178	1547	39679	1410
960	39544	1415	36210	1546	36194	1546	39655	1411
1280	39794	1406	36245	1544	36229	1545	39112	1431
1600	41180	1359	36244	1544	36209	1546	38210	1465
1920	42782	1308	36278	1543	36280	1543	38539	1452
2240	45412	1232	36330	1541	36363	1539	38299	1461
2560	46237	1210	36349	1540	36393	1538	38771	1444
2880	48346	1158	36361	1539	36504	1532	38384	1458
3200	52277	1070	36424	1537	36498	1533	38741	1445

Table 29: Runtime and throughput for evaluation of deduction rules for symmetry and transitivity of locally generated synthetic tree dataset (32 TripleWorker and 64 RequestWorker).

				serial		rounds		spinning		blocking single		blocking multi	
depth	breadth	triple	requests	time (ms)	Triple/s	time (ms)	Triple/s	time (ms)	Triple/s	time (ms)	Triple/s	time (ms)	Triple/s
1	2	9	3	14	642	216	41	61	147	41	219	14	642
2	2	49	7	24	2041	190	257	62	790	40	1225	16	3062
3	2	225	15	34	6617	196	1147	64	3515	51	1691	24	16271
4	2	961	31	51	18843	160	6006	65	14784	59	16288	41	23439
5	2	3969	63	158	25120	207	19173	110	36081	123	32268	81	49000
6	2	16129	127	1318	12237	630	25601	589	27383	511	31563	548	29432
7	2	65025	255	13061	4978	3989	16301	5069	12827	4399	14781	5002	12999
8	2	261121	511	120582	2165	29112	8969	34899	7482	35058	7448	34961	7468
9	2	1046529	1023	1053312	993	240034	4359	240717	4347	332670	3145	387940	2697

Table 30: Frequency and throughput in 1 000 repeated runs for retrieval of synthetic tree dataset and evaluation of deduction rules for symmetry and transitivity (32 TripleWorker and 64 RequestWorker).

		serial		rounds		spinning		blocking single		blocking multi		blocking multi reset	
depth	breadth	Hz	Triple/s	Hz	Triple/s	Hz	Triple/s	Hz	Triple/s	Hz	Triple/s	Hz	Triple/s
1	2	47	420	11	94	19	168	50	445	49	387	207	1860
2	2	42	2062	7	359	18	879	48	2356	48	2161	193	9448
3	2	29	6520	6	1256	16	3545	42	9437	43	9780	155	34829
4	2	13	12654	4	4175	12	11735	30	29202	32	31215	66	63503
5	2	4	16906	3	12234	7	27724	11	42861	11	45598	13	52805
6	2	0.7	10718	1	21858	2	30517	2	30621	2	32263	2	29655

Table 31: Runtime and throughput for retrieval with about 2 ms and 200 ms delay of synthetic tree dataset and evaluation of deduction rules for symmetry and transitivity (32 TripleWorker and 64 RequestWorker).

				serial		rounds		spinning		blocking single		blocking multi		
depth	breadth	triple	requests	time (ms)	Triple/s	time (ms)	Triple/s	time (ms)	Triple/s	time (ms)	Triple/s	time (ms)	Triple/s	delay (ms)
1	2	9	3	28	321	211	42	61	147	41	219	20	450	2
2	2	49	7	47	1042	140	349	62	790	41	1195	24	2041	2
3	2	225	15	68	3308	194	1159	83	2710	37	6081	27	8333	2
4	2	961	31	131	7335	247	3890	74	12986	42	22880	39	24641	2
5	2	3969	63	281	14124	359	11055	140	28349	110	36081	118	33635	2
6	2	16129	127	1684	9577	803	20085	615	26226	567	28446	641	25162	2
7	2	65025	255	13882	4684	4269	15231	3781	17197	4508	14424	4208	15452	2
8	2	261121	511	113566	2299	28986	9008	30181	8651	32723	7979	33540	7785	2
9	2	1046529	1023	972955	1075	218866	4781	281245	3721	252823	4139	304624	3435	2
1	2	9	3	626	14	501	17	453	19	435	20	419	21	200
2	2	49	7	1444	33	706	69	667	73	632	77	624	78	200
3	2	225	15	3067	73	931	241	859	261	832	270	1025	219	200
4	2	961	31	6320	152	1171	820	1068	899	1047	917	1227	783	200
5	2	3969	63	12896	307	1465	2709	1314	3020	1247	3182	1443	2750	200
6	2	16129	127	27032	596	2057	7841	1704	9465	1785	9035	2265	7120	200
7	2	65025	255	64506	1008	5237	12416	5161	12599	6156	10562	5187	12536	200
8	2	261121	511	217366	1201	30878	8456	29103	8972	33585	7774	42851	6093	200
9	2	1046529	1023	1201676	870	272087	3846	259398	4034	315747	3314	283315	3693	200

Table 32: Population, necessary manipulations and elapsed time in a Game of Life of size 10 x 10 over 100 generations.

gen.	pop.	death	birth	total manip.	elap. (ms)	gen.	pop.	death	birth	total manip.	elap. (ms)
1	11	37	11	48	159	51	5	5	5	10	50
2	15	15	15	30	139	52	8	1	8	9	44
3	9	9	9	18	109	53	7	14	7	21	59
4	9	8	9	17	114	54	11	7	11	18	57
5	9	5	9	19	82	55	10	12	10	22	80
6	7	14	7	21	95	56	2	3	2	5	35
7	10	7	10	17	106	57	3	1	3	4	32
8	9	10	9	19	101	58	3	5	3	8	39
9	15	9	15	24	102	59	6	4	6	10	34
10	10	20	10	30	118	60	6	9	6	15	43
11	15	11	15	26	111	61	12	2	12	14	57
12	14	14	14	28	87	62	6	20	6	26	68
13	9	10	9	19	91	63	2	8	2	10	40
14	10	11	10	21	65	64	2	2	2	4	36
15	12	7	12	19	64	65	2	2	2	4	35
16	14	17	14	31	130	66	2	2	2	4	32
17	15	12	15	27	103	67	2	2	2	4	33
18	7	15	7	22	85	68	2	2	2	4	32
19	11	8	11	19	69	69	2	2	2	4	27
20	10	11	10	21	81	70	2	2	2	4	38
21	9	8	9	17	58	71	2	2	2	4	26
22	10	3	10	13	72	72	2	2	2	4	27
23	10	14	10	24	99	73	2	2	2	4	24
24	14	10	14	24	93	74	2	2	2	4	34
25	10	19	10	29	81	75	2	2	2	4	29
26	7	7	7	14	48	76	2	2	2	4	29
27	5	5	5	10	46	77	2	2	2	4	30
28	7	3	7	10	43	78	2	2	2	4	31
29	7	8	7	15	82	79	2	2	2	4	34
30	9	9	9	18	72	80	2	2	2	4	32
31	9	7	9	16	61	81	2	2	2	4	30
32	10	10	10	20	68	82	2	2	2	4	33
33	12	12	12	24	74	83	2	2	2	4	39
34	12	12	12	24	62	84	2	2	2	4	27
35	14	8	14	22	81	85	2	2	2	4	31
36	11	21	11	32	103	86	2	2	2	4	35
37	9	14	9	23	72	87	2	2	2	4	29
38	9	8	9	17	48	88	2	2	2	4	30
39	8	5	8	13	74	89	2	2	2	4	29
40	9	8	9	17	61	90	2	2	2	4	27
41	10	5	10	15	57	91	2	2	2	4	32
42	10	7	10	17	65	92	2	2	2	4	28
43	9	13	9	22	77	93	2	2	2	4	29
44	7	13	7	20	71	94	2	2	2	4	38
45	15	8	15	23	60	95	2	2	2	4	23
46	10	20	10	30	75	96	2	2	2	4	34
47	10	8	10	18	53	97	2	2	2	4	30
48	5	9	5	14	62	98	2	2	2	4	31
49	4	6	4	10	42	99	2	2	2	4	31
50	4	3	4	7	38	100	2	2	2	4	28

Table 33: Population, necessary manipulations and elapsed time in a Game of Life of size 33 x 33 over 100 generations.

gen.	pop.	death	birth	total manip.	elap. (ms)
1	104	395	104	499	1566
2	121	129	121	250	673
3	102	115	102	217	483
4	101	106	101	207	415
5	107	98	107	205	438
6	118	107	118	225	378
7	119	105	119	224	368
8	109	124	109	233	358
9	128	117	128	245	399
10	100	143	100	243	414
11	108	90	108	198	339
12	99	111	99	210	337
13	110	103	110	213	361
14	89	121	89	210	353
15	105	94	105	199	317
16	94	112	94	206	340
17	100	78	100	178	276
18	77	113	77	190	138
19	68	93	68	161	185
20	67	75	67	142	260
21	62	72	62	134	150
22	65	58	65	123	180
23	60	71	60	131	206
24	66	61	66	127	240
25	56	77	56	133	125
26	61	49	61	110	99
27	53	66	53	119	106
28	58	58	58	116	109
29	60	48	60	108	130
30	73	53	73	126	292
31	62	67	62	129	219
32	65	57	65	122	146
33	56	73	56	129	166
34	75	64	75	139	102
35	69	63	69	132	87
36	70	79	70	149	162
37	75	83	75	158	201
38	64	70	64	134	221
39	60	64	60	124	158
40	49	65	49	114	91
41	57	45	57	102	51
42	51	62	51	113	64
43	52	52	52	104	59
44	54	47	54	101	49
45	54	46	54	100	50
46	57	57	57	114	55
47	55	56	55	111	177
48	62	57	62	119	194
49	66	59	66	125	130
50	49	66	49	115	116
51	58	53	58	111	122
52	45	72	45	117	82
53	57	47	57	104	74
54	46	51	46	97	61
55	50	41	50	91	81
56	43	47	43	90	63
57	50	57	50	107	139
58	53	44	53	97	132
59	38	45	38	83	104
60	50	34	50	84	137
61	48	53	48	101	94
62	55	62	55	117	167
63	59	51	59	110	146
64	58	67	58	125	215
65	56	52	56	108	106
66	63	57	63	120	49
67	59	64	59	123	188
68	48	57	48	105	129
69	50	41	50	91	109
70	52	58	52	110	135
71	48	48	48	96	134
72	47	52	47	99	110
73	45	38	45	83	120
74	40	44	40	84	110
75	45	56	45	101	97
76	46	39	46	85	83
77	49	63	49	112	119
78	38	36	38	74	69
79	48	31	48	79	38
80	45	52	45	97	41
81	36	54	36	90	35
82	37	34	37	71	32
83	32	29	32	61	71
84	40	39	40	79	50
85	33	50	33	83	71
86	40	30	40	70	37
87	32	44	32	76	43
88	29	24	29	53	23
89	28	33	28	61	43
90	28	31	28	59	28
91	31	28	31	59	31
92	24	17	24	41	22
93	24	36	24	60	31
94	35	23	35	58	24
95	24	29	24	53	30
96	24	24	24	48	27
97	27	22	27	49	24
98	32	23	32	55	32
99	33	35	33	68	28

Table 34: Population, necessary manipulations and elapsed time in a Game of Life of size 100 x 100 over 100 generations.

gen.	pop.	death	birth	total manip.	elap. (ms)	gen.	pop.	death	birth	total manip.	elap. (ms)
1	1076	3402	1076	4478	7774	51	434	418	434	852	418
2	1174	1429	1174	2603	3745	52	428	427	428	855	416
3	1104	1174	1104	2278	3130	53	466	440	466	906	432
4	1024	1184	1024	2208	1251	54	473	478	473	951	420
5	972	1003	972	1975	863	55	502	460	502	962	424
6	983	1004	983	1987	777	56	490	476	490	966	435
7	939	1044	939	1983	716	57	526	510	526	1036	472
8	961	896	961	1857	728	58	520	541	520	1061	484
9	873	979	873	1852	909	59	487	541	487	1028	458
10	909	899	909	1808	641	60	475	491	475	966	485
11	906	942	906	1848	648	61	463	463	463	926	431
12	853	874	853	1727	632	62	466	519	466	985	476
13	805	894	805	1699	758	63	460	452	460	912	456
14	768	815	768	1583	550	64	412	455	412	867	405
15	734	774	734	1508	557	65	404	418	404	822	393
16	760	735	760	1495	531	66	394	429	394	823	415
17	757	822	757	1579	567	67	420	365	420	785	389
18	733	822	733	1555	556	68	386	398	386	784	364
19	725	737	725	1462	528	69	391	393	391	784	382
20	752	753	752	1505	565	70	401	420	401	821	403
21	724	794	724	1518	539	71	350	385	350	735	361
22	716	744	716	1460	552	72	366	388	366	754	376
23	699	703	699	1402	494	73	384	361	384	745	383
24	650	721	650	1371	507	74	366	427	366	793	378
25	683	636	683	1319	495	75	349	316	349	665	345
26	629	726	629	1355	504	76	361	409	361	770	356
27	615	609	615	1224	514	77	336	307	336	643	386
28	587	613	587	1200	519	78	398	355	398	753	430
29	548	592	548	1140	512	79	377	372	377	749	376
30	582	583	582	1165	510	80	389	338	389	727	395
31	537	600	537	1137	504	81	356	368	356	724	399
32	514	497	514	1011	461	82	362	386	362	748	390
33	511	559	511	1070	492	83	396	347	396	743	376
34	537	525	537	1062	477	84	384	347	384	731	386
35	524	522	524	1046	476	85	407	419	407	826	407
36	496	485	496	981	448	86	409	434	409	843	429
37	498	545	498	1043	472	87	370	417	370	787	416
38	483	523	483	1006	440	88	383	364	383	747	426
39	432	501	432	933	413	89	364	389	364	753	428
40	435	451	435	886	415	90	378	401	378	779	370
41	421	449	421	870	408	91	339	361	339	700	407
42	406	427	406	833	412	92	317	358	317	675	412
43	392	363	392	755	354	93	351	324	351	675	392
44	433	445	433	878	406	94	341	384	341	725	393
45	429	409	429	838	403	95	312	323	312	635	390
46	444	465	444	909	427	96	340	357	340	697	382
47	439	425	439	864	420	97	293	294	293	587	396
48	433	464	433	897	428	98	301	307	301	608	407
49	434	426	434	860	391	99	279	275	279	554	415
50	440	423	440	863	389	100	299	305	299	604	395

A.3 RULE PROGRAM EXAMPLES

Listing 19: Custom rule set for LUBM.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix: lubm <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>.

 { ?x rdf:type lubm:Program. } => { ?x rdf:type lubm:Organization. }.
 { ?x rdf:type lubm:PostDoc. } => { ?x rdf:type lubm:Faculty. }.
 { ?x lubm:title ?y. } => { ?x rdf:type lubm:Person. }.
 { ?x rdf:type lubm:Dean. } => { ?x rdf:type lubm:Professor. }.
 { ?y lubm:subOrganizationOf ?x. } => { ?x rdf:type lubm:Organization. }.
 { ?y lubm:orgPublication ?x. } => { ?y rdf:type lubm:Organization. }.
 { ?y lubm:softwareVersion ?x. } => { ?y rdf:type lubm:Software. }.
 { ?y lubm:undergraduateDegreeFrom ?x. } => { ?y rdf:type lubm:Person. }.
 { ?y lubm:member ?x. } => { ?y rdf:type lubm:Organization. }.
 { ?x lubm:emailAddress ?y. } => { ?x rdf:type lubm:Person. }.
 { ?x rdf:type lubm:Institute. } => { ?x rdf:type lubm:Organization. }.
 { ?y lubm:teacherOf ?x. } => { ?x rdf:type lubm:Course. }.
 { ?y lubm:softwareDocumentation ?x. } => { ?y rdf:type lubm:Software. }.
 { ?y lubm:researchProject ?x. } => { ?x rdf:type lubm:Research. }.
 { ?x rdf:type lubm:Book. } => { ?x rdf:type lubm:Publication. }.
 { ?x rdf:type lubm:Chair. } => { ?x rdf:type lubm:Professor. }.
 { ?y lubm:advisor ?x. } => { ?x rdf:type lubm:Professor. }.
 { ?y lubm:teachingAssistantOf ?x. } =>
                  { ?y rdf:type lubm:TeachingAssistant. }.
 { ?y lubm:affiliateOf ?x. } => { ?x rdf:type lubm:Person. }.
 { ?x rdf:type lubm:ResearchGroup. } => { ?x rdf:type lubm:Organization. }.
 { ?x rdf:type lubm:Manual. } => { ?x rdf:type lubm:Publication. }.
 { ?x rdf:type lubm:Faculty. } => { ?x rdf:type lubm:Employee. }.
 { ?x rdf:type lubm:Article. } => { ?x rdf:type lubm:Publication. }.
 { ?x lubm:worksFor ?y. } => { ?x lubm:memberOf ?y. }.
 { ?y lubm:member ?x. } => { ?x rdf:type lubm:Person. }.
 { ?y lubm:advisor ?x. } => { ?y rdf:type lubm:Person. }.
 { ?y lubm:undergraduateDegreeFrom ?x. } => { ?x rdf:type lubm:University. }.
 { ?x rdf:type lubm:VisitingProfessor. } => { ?x rdf:type lubm:Professor. }.
 { ?x rdf:type lubm:University. } => { ?x rdf:type lubm:Organization. }.
 { ?x rdf:type lubm:Research. } => { ?x rdf:type lubm:Work. }.
 { ?y lubm:listedCourse ?x. } => { ?y rdf:type lubm:Schedule. }.
 { ?x rdf:type lubm:Department. } => { ?x rdf:type lubm:Organization. }.
 { ?y lubm:affiliatedOrganizationOf ?x. } =>
                  { ?x rdf:type lubm:Organization. }.
 { ?x rdf:type lubm:Software. } => { ?x rdf:type lubm:Publication. }.
 { ?y lubm:doctoralDegreeFrom ?x. } => { ?x rdf:type lubm:University. }.
 { ?x rdf:type lubm:Professor. } => { ?x rdf:type lubm:Faculty. }.
 { ?x rdf:type lubm:AssistantProfessor. } => { ?x rdf:type lubm:Professor. }.
 { ?x lubm:subOrganizationOf ?y.
   ?y lubm:subOrganizationOf ?z. } => { ?x lubm:subOrganizationOf ?z. }.
 { ?x rdf:type lubm:ClericalStaff. } =>
                  { ?x rdf:type lubm:AdministrativeStaff. }.
 { ?x rdf:type lubm:SystemsStaff. } =>
                  { ?x rdf:type lubm:AdministrativeStaff. }.
 { ?y lubm:orgPublication ?x. } => { ?x rdf:type lubm:Publication. }.
 { ?y rdf:type lubm:Department.
   ?x lubm:headOf ?y. } => { ?x rdf:type lubm:Chair. }.
 { ?y rdf:type lubm:College.
   ?x lubm:headOf ?y. } => { ?x rdf:type lubm:Dean. }.
 { ?x rdf:type lubm:JournalArticle. } => { ?x rdf:type lubm:Article. }.
 { ?x lubm:doctoralDegreeFrom ?y. } => { ?x lubm:degreeFrom ?y. }.
 { ?x rdf:type lubm:Lecturer. } => { ?x rdf:type lubm:Faculty. }.
 { ?y lubm:doctoralDegreeFrom ?x. } => { ?y rdf:type lubm:Person. }.
 { ?y lubm:publicationDate ?x. } => { ?y rdf:type lubm:Publication. }.
 { ?y lubm:mastersDegreeFrom ?x. } => { ?x rdf:type lubm:University. }.
 { ?x lubm:mastersDegreeFrom ?y. } => { ?x lubm:degreeFrom ?y. }.
 { ?y lubm:degreeFrom ?x. } => { ?x lubm:hasAlumnus ?y. }.
```

```
{ ?x lubm:age ?y. } => { ?x rdf:type lubm:Person. }.
{ ?x rdf:type lubm:AssociateProfessor. } => { ?x rdf:type lubm:Professor. }.
{ ?x rdf:type lubm:Course. } => { ?x rdf:type lubm:Work. }.
{ ?y lubm:hasAlumnus ?x. } => { ?y rdf:type lubm:University. }.
{ ?x rdf:type lubm:ResearchAssistant. } => { ?x rdf:type lubm:Student. }.
{ ?y lubm:listedCourse ?x. } => { ?x rdf:type lubm:Course. }.
{ ?x rdf:type lubm:TechnicalReport. } => { ?x rdf:type lubm:Article. }.
{ ?x rdf:type lubm:FullProfessor. } => { ?x rdf:type lubm:Professor. }.
{ ?x rdf:type lubm:ConferencePaper. } => { ?x rdf:type lubm:Article. }.
{ ?x rdf:type lubm:UnofficialPublication. } =>
                { ?x rdf:type lubm:Publication. }.
{ ?x lubm:takesCourse ?y.
  ?y rdf:type lubm:Course. } => { ?x rdf:type lubm:Student. }.
{ ?y lubm:hasAlumnus ?x. } => { ?x lubm:degreeFrom ?y. }.
{ ?y lubm:publicationResearch ?x. } => { ?y rdf:type lubm:Publication. }.
{ ?x lubm:headOf ?y. } => { ?x lubm:worksFor ?y. }.
{ ?y lubm:softwareDocumentation ?x. } => { ?x rdf:type lubm:Publication. }.
{ ?y lubm:subOrganizationOf ?x. } => { ?y rdf:type lubm:Organization. }.
{ ?y lubm:affiliatedOrganizationOf ?x. } =>
                { ?y rdf:type lubm:Organization. }.
{ ?y lubm:affiliateOf ?x. } => { ?y rdf:type lubm:Organization. }.
{ ?y lubm:publicationAuthor ?x. } => { ?y rdf:type lubm:Publication. }.
{ ?y rdf:type lubm:Organization.
  ?x lubm:worksFor ?y. } => { ?x rdf:type lubm:Employee. }.
{ ?y lubm:tenured ?x. } => { ?y rdf:type lubm:Professor. }.
{ ?x lubm:undergraduateDegreeFrom ?y. } => { ?x lubm:degreeFrom ?y. }.
{ ?y lubm:degreeFrom ?x. } => { ?x rdf:type lubm:University. }.
{ ?y lubm:researchProject ?x. } => { ?y rdf:type lubm:ResearchGroup. }.
{ ?y lubm:mastersDegreeFrom ?x. } => { ?y rdf:type lubm:Person. }.
{ ?y lubm:degreeFrom ?x. } => { ?y rdf:type lubm:Person. }.
{ ?x rdf:type lubm:Specification. } => { ?x rdf:type lubm:Publication. }.
{ ?x lubm:memberOf ?y. } => { ?y lubm:member ?x. }.
{ ?x rdf:type lubm:GraduateCourse. } => { ?x rdf:type lubm:Course. }.
{ ?x rdf:type lubm:GraduateStudent. } => { ?x rdf:type lubm:Person. }.
{ ?x rdf:type lubm:AdministrativeStaff. } => { ?x rdf:type lubm:Employee. }.
{ ?y lubm:hasAlumnus ?x. } => { ?x rdf:type lubm:Person. }.
{ ?x lubm:teachingAssistantOf ?y.
  ?y rdf:type lubm:Course.
  ?x rdf:type lubm:Person. } => { ?x rdf:type lubm:TeachingAssistant. }.
{ ?y lubm:publicationAuthor ?x. } => { ?x rdf:type lubm:Person. }.
{ ?y lubm:teachingAssistantOf ?x. } => { ?x rdf:type lubm:Course. }.
{ ?x rdf:type lubm:College. } => { ?x rdf:type lubm:Organization. }.
{ ?x lubm:telephone ?y. } => { ?x rdf:type lubm:Person. }.
{ ?y lubm:publicationResearch ?x. } => { ?x rdf:type lubm:Research. }.
{ ?y rdf:type lubm:Program.
  ?x lubm:headOf ?y. } => { ?x rdf:type lubm:Director. }.
{ ?x rdf:type lubm:UndergraduateStudent. } => { ?x rdf:type lubm:Student. }.
{ ?x lubm:member ?y. } => { ?y lubm:memberOf ?x. }.
{ ?y lubm:teacherOf ?x. } => { ?y rdf:type lubm:Faculty. }.
```

Listing 20: Rules for Game of Life.

```
@prefix http: <http://www.w3.org/2011/http#> .
@prefix httpm: <http://www.w3.org/2011/http-methods#> .
@prefix gol: <http://localhost:8888/gameoflife/> .

{ ?cell gol:state "alive".
  ?cell gol:livingNeighbors "4". }=>{ ?cell gol:change "death". }.

{ ?cell gol:state "alive".
  ?cell gol:livingNeighbors "5". }=>{ ?cell gol:change "death". }.

{ ?cell gol:state "alive".
  ?cell gol:livingNeighbors "6". }=>{ ?cell gol:change "death". }.

{ ?cell gol:state "alive".
```

```
15      ?cell gol:livingNeighbors "7". }=>{ ?cell gol:change "death". }.
16
17    { ?cell gol:state "alive".
18      ?cell gol:livingNeighbors "8". }=>{ ?cell gol:change "death". }.
19
20    { ?cell gol:state "alive".
21      ?cell gol:livingNeighbors "1". }=>{ ?cell gol:change "death". }.
22
23    { ?cell gol:state "alive".
24      ?cell gol:livingNeighbors "0". }=>{ ?cell gol:change "death". }.
25
26    { ?cell gol:state "dead".
27      ?cell gol:livingNeighbors "3". }=>{ ?cell gol:change "birth". }.
28
29    { ?cell gol:change "death". } => { [] http:mthd httpm:POST ;
30                                          http:requestURI ?cell ;
31                                          http:body { ?cell gol:state "dead". }.
32                                     }.
33
34    { ?cell gol:change "birth".} => { [] http:mthd httpm:POST ;
35                                        http:requestURI ?cell ;
36                                        http:body { ?cell gol:state "alive". }.
37                                   }.
```