Increasing the number of open data streams on the Web

Bachelor Thesis in Business Informatics

Department of Informatics
University of Zurich
Prof. Abraham Bernstein, Ph.D.

Universität Zürich[UZH]

Advisor
Dr. Daniele Dell'Aglio

Author
Patrick Muntwyler
Wohlen, AG, Switzerland
Student-ID 14-711-337

Zurich, 26.07.2017

Acknowledgments

I would like to thank Dr. Daniele Dell'Aglio for his support during this thesis. I appreciate the time he took for all the meetings. I am grateful that I had the chance to work with him and learn from him. He showed me what it means to write a thesis and that it can also be fun. I am thankful for his inputs, recommendations and advices. Also, I would like to thank him for reviewing my thesis.
I am grateful for the support of my friends. They motivated me and always had time to talk when I needed it. Especially, I would like to thank Timon and Christoph for their help.
Finally, I am truly grateful for the help provided by my family. Because of their support I was able to fully concentrate on working on my thesis. They made it possible for me to work under the best conditions.

Abstract

Open data describes data which are available for everyone and often can be used for any purpose. The term open data does not only stand for this type of data, but also for the movement which supports it. The idea behind open data is to create more transparency and other advantages for society. Especially the data owned by governments can increase the quality of living when made accessible to the public. The associated term is called Open Government Data. Open Government Data can increase the public confidence and enables the development of new services which use these Open Government Data to ease the everyday life of citizens. Thus in recent times, the number of governments increases, which support this trend and publish their data.

Linked Data was invented to increase the value of data, which are published on the Web. Linked Data uses a specific format, which enables to link data from different data sources. Therefore, machine processes can gather data from different sources, which can lead to better results. TripleWave is a framework, that applies the concept of Linked Data to streaming data. TripleWave can transform existing streams into Linked Data streams.

However, it lacks a prototype, which shows how TripleWave can be used to publish Open Government Data as Linked Data streams. Such an approach would combine the advantages and characteristics of the areas Open Government Data, Linked Data and streaming data.

In the context of this Bachelor Thesis we increase the number of open data streams on the Web. First, we examine the available Open Government Data portals and search for data sets that are suitable to be streamed on the Web. Only data sets, which are updated frequently and have a time reference, are suitable to be transformed and streamed. Then we develop an application, which fetches and transforms several Open Government Data sets and finally publishes them as Linked Data streams on the Web.

Kurzfassung

Open Data steht für Daten, die für alle zugänglich sind und oft für jegliche Zwecke benutzt werden dürfen. Der Begriff Open Data steht jedoch nicht nur für diese Art von Daten, sondern auch für die dahinterstehende Bewegung, die Open Data unterstützt. Die Idee hinter Open Data ist, dass mehr Transparenz und weitere Vorteile für die Zivilgesellschaft geschaffen werden. Besonders Regierungen besitzen Daten, die die Lebensqualität der Einwohner erhöhen kann, wenn sie veröffentlicht werden. Der dazugehörende Begriff heisst Open Government Data. Open Government Data kann das öffentliche Vertrauen in die Regierung verbessern und ermöglicht die Entwicklung von neuen Dienstleistungen, die auf diesen Daten beruhen und das Alltagsleben der Einwohner vereinfachen. In jüngster Zeit steigt daher die Anzahl an Regierungen, die diesen Trend unterstützen und ihre Daten veröffentlichen.

Linked Data wurde entwickelt um den Wert von Daten zu erhöhen, die im Internet veröffentlicht werden. Linked Data benutzt ein spezifisches Format, das es ermöglicht, Daten aus verschiedenen Quellen zu verbinden. Dadurch wird es maschinellen Prozessen ermöglicht, Daten aus verschiedenen Quellen zu benutzen, um bessere Resultate zu erzielen. TripleWave ist ein Framework, welches das Konzept von Linked Data auf Streaming-Daten anwendet. TripleWave kann existierende Streams in Linked Data Streams transformieren.

Es fehlt jedoch ein Prototyp, der zeigt, wie man mit TripleWave Open Government Data als Linked Data Streams veröffentlichen kann. Ein solcher Ansatz würde die Vorteile und Charakteristiken aus den Gebieten Open Government Data, Linked Data und Streaming-Daten vereinen.

Im Rahmen dieser Bachelorarbeit erhöhen wir die Anzahl von Open Data Streams im Web. Zuerst untersuchen wir die verfügbaren Open Government Data Portale und suchen nach Datensets, die sich dazu eignen, als Streams im Web veröffentlicht zu werden. Nur Datensets, die häufige Änderungen erfahren und eine zeitliche Komponente beinhalten, eignen sich transformiert und gestreamt zu werden. Dann entwickeln wir eine Applikation, die mehrere Open Government Datensets abholt, transformiert und sie zum Schluss im Web als Linked Data Streams veröffentlicht.

Contents

List of Figures

List of Tables

1 Introduction

The open data movement[1] expends effort for spreading the concepts of open data and open knowledge. Open data defines data which are available for everyone and which often has no restrictions concerning its usage. Open knowledge is created when open data is appropriately used. The members of this movement want the people to be aware of the advantages of open data for the civil society. They help organisations and governments to publish and use open data so that the benefit for the society is as big as possible. Their vision is a world where open knowledge is an essential element in the life of everyone.

An important institution for open data are governments as they own a lot of data and information about structures, processes and inhabitants of countries or municipalities. If these data and information are made available to the community, new services can be generated and the quality of living can increase. The open data movement has a Working Group on Open Government Data[2] which supports the publication of government data.

There is a lot of progress in the field of Open Government Data. Several countries and big cities publish their data on open data portals. For example, there are open data portals available for the city of Zurich[3], Switzerland[4], Germany[5] and Austria[6], mentioning some portals of the German-language area.

To increase the value of data, one can publish data sets following the Linked Data Principles (Bizer, Heath & Berners-Lee 2009). The Linked Data Principles describe an approach of how data sets can be interlinked in the Web using a format called Resource Description Framework (RDF) (Cyganiak, Wood & Lanthaler 2014). Linking data across different data sources according to the Linked Data Principles enables machine processes to query several data sets as in a distributed database. Thus the results of these machine processes can contain more precise information. One of the biggest achievements of the Linked Data movement is the Linking Open Data cloud[7] where more than one thousand open data sets are published and interlinked, also including Open Government Data portals.

Many organisations and research projects use the Linked Data approach to increase their value or utility. Shadbolt et al. (2012) apply the Linked Data Principles to several related data sets from the open data portal of the United Kingdom. Their goal is to increase the value of those data sets and make them available through an application which eases the analyzation and visualization of the data. A non-profit project which applies the Linked Data Principles is Wikidata (Vrandencic & Krötzsch 2014). The entities which are stored in Wikidata have unique identifiers. Thus other data sets can link to entities on Wikidata. Kobilarov et al. (2009) use the Linked Data Principles to reorganize and increase the informative content of the BBC Web sites.

Current efforts target mainly static data. But as the amount of data which are daily produced drastically increases, the need for data streams grows. Data streams are transient and no persistent storage is needed for processing. Especially as sensor data and the Internet of Things become popular, the need for transient data processing increases.

---

[1] https://okfn.org/about/ (accessed 11.7.2017)

[2] https://opengovernmentdata.org/ (accessed 11.7.2017)

[3] https://data.stadt-zuerich.ch/ (accessed 11.7.2017)

[4] https://opendata.swiss/en/ (accessed 11.7.2017)

[5] https://www.govdata.de/ (accessed 11.7.2017)

[6] https://www.data.gv.at/ (accessed 11.7.2017)

[7] http://lod-cloud.net/ (accessed 11.7.2017)

W3C has a community group which works on defining and processing Linked Data streams[8]. There are efforts made to enhance the publication of Linked Data as streams and to continuously querying them. Barbieri & Della Valle (2010) present a proposal about how to publish Linked Data as streams. In contrast to other papers which cover the topic of Linked Data streams, they propose an approach which considers the characteristics of data streams, especially that streaming data are transient and not persistent.

Mauri et al. (2016) have built a framework called TripleWave, based on the proposal of Barbieri & Della Valle (2010) and real-world use cases. TripleWave can transform non-RDF streams and RDF data sets which contain time stamps into RDF streams. The resulting output stream can easily be consumed through pull-based and push-based mechanisms. Mauri et al. (2016) have thus created a simply usable framework for publishing Linked Data streams.

But there are no guidelines and prototypes which show how open data sets can be published as Linked Data streams using TripleWave. Such guidelines would simplify the work for other studies in the field of Linked Open Data streams. A prototype would show how the steps can be implemented which are needed for fetching, transforming and publishing open data sets as RDF streams.

The goal of this project is to increase the number of Linked Open Data streams on the Web (using TripleWave). We present the challenges of our implementation and show how we met them. The result of our work is a prototype which combines TripleWave and Apache Kafka[9], a framework which is used for streaming activities in several big companies. Combining these two frameworks enables us to create a scalable and modular prototype, which can easily be extended. The cluster fetches non-RDF open data sets through pull-based mechanisms, transforms them and finally publishes them as RDF streams through push-based mechanisms.

The development of this thesis is split into three parts. In the first part, we survey open data sets which are suitable to be transformed into streams. We focus on German-language open data portals, and in particular, the open data portals of Zurich and of Switzerland. The survey is then complemented by data sets from Swiss public transportation portals and from examples from the open data portals of Germany and Austria.

The second part consists of implementing the needed components for our prototype. We need components which fetch the data sets and feed our cluster. Then components are required to connect Kafka and TripleWave. We use them also to publish the final RDF streams. For the transformation of the open data sets into RDF streams, we need specific mappings, one for each data set. Finally, we mention some rudimentary example client applications which consume the RDF streams. We publish the output streams through three different mechanisms: WebSockets (Fette & Melnikov 2011), Server Sent Events (Hickson 2015) and Apache Thrift[10].

In the last part of this thesis we evaluate our prototype. We measure the latency of our prototype in different setups and measure the processing times. We study the scalability by measuring the throughput for different setups. Finally, we compare the package sizes of output stream objects for different push-based mechanisms.

The thesis is structured as follows: in Chapter 2 we present the related work. In Chapter 3 we first present the 5-star open data scheme[11], which describes the different levels of open data. Then we describe how we create the survey of available open data sets which are suitable for being transformed into RDF streams. Afterwards we present the results of our survey. Chapter 4 presents the implemented components, introduces the mappings, which we created to transform the data into Linked Data and finally

[8] https://www.w3.org/community/rsp/ (accessed 11.7.2017)

[9] https://kafka.apache.org/ (accessed 11.7.2017)

[10] https://thrift.apache.org/ (accessed 23.7.2017)

[11] https://www.w3.org/DesignIssues/LinkedData.html (accessed 11.7.2017)

describes the prototype which we run on the University server. In Chapter 5 we present our evaluation results. Finally, Chapter 6 contains the conclusions and some final remarks.

2 Related Work

In this section we present the related work. We begin with an introduction into open data and Open Government Data. After that, we explain the term Linked Data and illustrate it with some examples. This is followed by a passage describing projects that implement Linked Data streams, including the framework TripleWave, which is an important part of this thesis. Subsequently the framework Apache Kafka is introduced followed by a section about Apache Thrift. Both, Apache Kafka and Apache Thrift, are used in this work.

2.1 Open Data

The term open is described according to Open Knowledge International[12]. Intuitively, something that is open, must be published under an *open license* or must be available in public domain. Public domain means that there is no copyright or similar restrictions. An open license contains the following conditions: A resource, that is provided under an open license must be free to be used, redistributed and modified. This must also hold for only parts of that artefact or in some cases if it is merged with other things that are published under an open license (e.g. under a share-alike license[13]). It must be ensured that the license must not discriminate against any group or person. Something that is open must be available for everyone. It must be available for no charge and in most cases for any purpose (a non-commercial license forbids the use of the artefacts for commercial intention[14]). There are some acceptable conditions an open license can make, for example that something that is published under an open license should include the attribution of contributors, right holder, sponsors and creators or that derivated artefacts of it must also happen under the same or a similar license. A detailed definition of the term *open* can be found here[15].

Open data refers to data that are published under conditions that fulfil the requirements of the term open: "Open data are the building blocks of open knowledge. Open knowledge is what open data becomes when it is useful, usable and used" (Open Knowledge International 2017a). As a summary, we can say that knowledge "is open if anyone is free to access, use, modify, and share it — subject, at most, to measures that preserve provenance and openness" (Open Knowledge International 2017b). The same summary can be applied to open data because, as mentioned before, knowledge is nothing else as applied data.

Open Government Data (OGD) are data that are "produced or commissioned by government or government controlled entities" (Open Knowledge International 2017c) and fulfils the conditions defined by the term open. Ideally, OGD creates transparency towards the citizens. Citizens can access, share and reuse OGD and analyse what the government is doing. Additionally, this data contains a lot of social and commercial value. By making this data accessible to the citizens, new and innovative applications can be implemented, to create social and commercial value (Open Knowledge International 2017c).

A good example for OGD are the urban sensor data streams described by Boyle, Yates & Yeatman (2013). Cities steadily grow and attract more people to try their luck for a good life in a city. To be attractive for companies and smart people the cities must proceed scaling effectively. This goal can be supported by information and communication technologies (ICT). But ICT must be fed with suitable data so that they can contribute a positive development of a city by offering new and enhanced city services. Boyle, Yates & Yeatman (2013) examine the available urban sensor data streams in London

[12] http://opendefinition.org/od/2.1/en/ (accessed 11.7.2017)

[13] https://creativecommons.org/licenses/by-sa/4.0/ (accessed 11.7.2017)

[14] https://creativecommons.org/licenses/by-nc/4.0/ (accessed 11.7.2017)

[15] http://opendefinition.org/od/2.1/en/ (accessed 31.5.2017)

in 2013 to provide better understanding for these streams so that they will be used by a broader community.

Most of the examined urban sensor data streams origin from the mass transit and environment sectors. Public transportation in London are subject to the Mayor's Office with Transport of London (TfL) which is responsible for most aspects. TfL is under the 2000 Freedom of Information Act and so most of the information owned by TfL is made available to the public. One example is the information about London's Underground network, which is collected by the TrackerNet system. TfL worked together with Microsoft to implement an open real-time data feed including information like train prediction, station status or line status. London's iBus system is another example for publishing real-time data by the government. London's iBus system includes data about each bus: speed, direction, location and more. One specific service which is built on London's Open Transport Data, called Citymapper, helps to ease the life of London's public transport users (Scott 2015). Concerning the environmental sector, London has open data streams available with data about weather observations and forecasts, London air quality or information about water, like river levels or water quality generally (Boyle, Yates & Yeatman 2013).

Boyle, Yates & Yeatman (2013) mention that a lot of new open data streams will be implemented in future. There is an initiative to measure Heathrow Airport's air quality. There are also plans from the bureau of the mayor to implement sensing systems concerning the water supply system. Therefore, leaks in pipes could be easily observed or the water quality could be measured. Boyle, Yates & Yeatman (2013) also mention Santander, a city in Spain, which was named Europe's Smartest City in 2013, largely due to the high number of sensors implemented in the city, monitoring a lot of things like environmental or parking conditions.

2.2 Linked Data

Linked Open Data are the next level of open data. So we explain in the first part of this chapter the term Linked Data. Afterwards the Linking Open Data Project is shortly presented, followed by some projects which have implemented Linked Data on its top.

Bizer, Heath & Berners-Lee (2009) explain that Linked Data are data which are scattered over the Web and there exist typed links between these distributed data sources. This is a contrast to the Hypertext Web which uses untyped links to link Websites. Linked Data characteristics are: the data are published in a machine-readable format; the content and its sense are defined explicitly; the data contain typed links to other data or are referenced by other data. Documents which contain Linked Data use the Resource Description Framework (RDF) (Cyganiak, Wood & Lanthaler 2014). We give a short introduction into RDF below. Instead of linking such documents with untyped links, Linked Data uses RDF for creating typed links. These typed links can link data about any entities in the world. As a result of this approach, the so called Web of Data has been formed.

Berners-Lee (2006) defined a guideline for publishing Linked Data. This guideline includes the following four rules which are also known as the Linked Data Principles:

- "Use URIs as names for things"
- "Use HTTP URIs so that people can look up those names"
- "When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)"
- "Include links to other URIs. [sic] so that they can discover more things"

Bizer, Heath & Berners-Lee (2009) mention that Linked Data are based on two technologies which are crucial for the Web: Uniform Resource Identifiers (URIs) (Berners-Lee, Fielding & Masinter 2005) and the HyperText Transfer Protocol (HTTP) (Fielding et al. 1999). More people are familiar with the term URL than with URI. The difference between these two terms is that an URL is an address for documents and other entities on the Web while an URI can be used as an ID for anything. Thus also real world entities can be identified by URIs. If URIs are used in combination with the HTTP Protocol, then this

URI can be dereferenced and information about the entity which is identified by the corresponding URI can be retrieved.

Both, URIs and HTTP are important parts of the RDF format. RDF is a graph-based data model that can be used to link data and to define relations between data that give a description of entities in the world. The RDF Format consists of triples: Subject – Predicate – Object. Subjects are always URIs which point to an entity. The predicate is a relation between subject and the object and it is denoted by a URI. The object can be a URI, which points to another entity or it can be a string literal, which represents a value (Bizer, Heath & Berners-Lee 2009).

Bizer, Heath & Berners-Lee (2009, p. 4) give two examples for an RDF triple:

"Subject: *http://dig.csail.mit.edu/data#DIG*"
"Predicate: *http://xmlns.com/foaf/0.1/member*"
"Object: *http://www.w3.org/People/Berners-Lee/card#i*"

"Subject: *http://data.linkedmdb.org/resource/film/77*"
"Predicate: *http://www.w3.org/2002/07/owl#sameAs*"
"Object: *http://dbpedia.org/resource/Pulp_Fiction_%28film%29*"

In the first example the subject is the URI *http://dig.csail.mit.edu/data#DIG*, which can be dereferenced and shows a description of the MIT Decentralized Information Group. The object is also an URI, which points to the entity Tim Berners-Lee. The predicate is an URI which defines the link type *member*. The statement of this triple is that Tim Berners-Lee is a member of the MIT Decentralized Information Group. The second triple contains as subject and object two URIs which point to the movie Pulp Fiction in two different data bases. The predicate is the URI *http://www.w3.org/2002/07/owl#sameAs* which is used to create a link between these two entities in these two data bases. The predicate says that both entities describe the same real-world entity. For example, a search engine can crawl one of these two data bases and it will find a link to the other data base which means that it will be able to aggregate the data from both data bases.

Bizer, Heath & Berners-Lee (2009) explain that vocabularies describe real-world entities and the relations between them. Vocabularies can be designed by using the RDF Vocabulary Definition Language (RDFS) (Brickley & Guha 2014) and the Web Ontology Language (OWL) (McGuinness & van Harmelen 2004). They allow defining classes and properties in RDF. Vocabularies can be linked to other vocabularies by using RDF to describe the relation among each.

Bizer, Heath & Berners-Lee (2009) mention the Linking Open Data (LOD) Project which has been launched by the W3C Sematic Web Education and the Outreach Group in 2007. The goal of this project was to initiate the Web of Data. This was achieved by applying the Linked Data Principles to open licensed data bases. The result is a huge network that links data from many different data bases. The network has heavily grown since 2007. The graph of this network is too big for being printed properly on one page but can be viewed here[16]. The big growth can be ascribed to the nature of Linked Data. Everyone can help to increase the Web of Data by publishing open data following the Linked Data Principles.

One example of broaden the Linked Data Web (LDW) is the project initialized by Shadbolt et al. (2012). They took OGD sets from the United Kingdom's (UK) OGD portal, called data.gov.uk, and have applied the Linked Data Principles to them. On data.gov.uk thousands of data sets are published. Most of them are available as CSV file or as spreadsheets. OGD sets are normally of high quality and thus are a good choice for using them to broaden the LDW by transforming them into RDF.

According to Shadbolt et al. (2012) the process of converting OGD to RDF contains four important research challenges. The first one is to find data sets which are useful for applications. There are services

[16] http://lod-cloud.net/ (accessed 05.06.2017)

which can help to find public sector information (PSI), like the data.gov.uk portal. But to create innovative usage of PSI, sometimes there are data sets needed for example from different nations. This could be the case when using meteorological data. But there are no such services which easily find related data sets from different nations or different sources at all.

After finding useful data sets the second challenge is to integrate them into the LDW. Shadbolt et al. (2012) have chosen six data sets. The goal is to link these data sets to generate more information. To achieve that it is necessary to convert them to RDF. Shadbolt et al. (2012) recommend to use popular ontologies, to reduce the effort of modelling as much as possible.

Once the ontologies are defined, the third challenge is to find the best join points (where the data can be linked) in the distinct data sets. They decided to link the data sets over geographical similarities, because on the one hand the LDW already contains a lot of geographical resources and on the other hand PSI data often has a geographical dimension, as for example the allocation of a crime rate to a district.

The last challenge is to build an application, which a user can use to analyse and visualise the data. Important for such an application is that the user can interact with it without having a lot of effort or requiring a lot of knowledge in programming. With such an approach, more people are motivated to use that application and to deal with the data.

A big project which also implements the Linked Data Principles is Wikidata. Vrandencic & Krötzsch (2014) describe what Wikidata is and how it can influence the opportunities for many new applications. Wikipedia contains more than 30 million articles in 287 languages. Thus it would be quite elaborate to extract the needed data out of it. This is where Wikidata comes into play. Wikidata aims to create new solutions of how this huge amount of data can be managed in a consistent way and therefore make it easily accessible for users. Wikidata is influenced by the following design characteristics: Open Editing, Community Control, Plurality, Secondary Data, Multilingual Data, Easy Access and Continuous Evolution.

There are other free knowledge base projects available in the Web, but they all suffer from certain disadvantages compared to Wikidata. For example, Semantic MediaWiki is not able to create a knowledge base for several languages which can include all Wikimedia projects. OpenCyc, another example, cannot be edited by the publicity. DBPedia and Yago extract data directly from the Wikipedia pages. These extracted data however are used in other projects like the "Google Knowledge Graph, Facebook's Open Graph, Wolfram Alpha Evi and IBM's Watson" (Vrandencic & Krötzsch 2014, p. 80). Thus all these applications would benefit from up-to-date and machine-readable data exports which will be enabled by Wikidata (Vrandencic & Krötzsch 2014).

Vrandencic & Krötzsch (2014) mention a pleasant trend of the community's people using external identifiers to link Wikidata items with objects of external data bases. In this way, the Wikidata entities can be enriched by additional data. Every Wikidata entity has its own identifier in form of a URI. This URI is resolvable and enables to fetch the corresponding item. This approach fulfils the Linked Data Principles. Thus Wikidata extends the LDW.

A further project which uses Linked Data to increase the value of an application, is the one of Kobilarov et al. (2009). They worked together with BBC to make their online content more attractive. Kobilarov et al. (2009) note that a user can find online a lot of content from BBC about several topics. There are domain-specific microsites for several themes like news, music, food and programmes. The problem is that the microsites are isolated among themselves and thus the information is not linked. For example, one can look up who moderates a certain TV show, but it is not shown which other TV shows this person has presented. The goal of that project is to connect the available information on the microsites to increase the potential of the BBC resources.

Kobilarov et al. (2009) mention that BBC has a legacy auto-categorization system called CIS, which was used to add annotations to local news. It contains terms about "proper names, subjects, brands, time periods and places" (Kobilarov et al. 2009, p. 727). They decided to use CIS for annotating the BBC programmes. CIS covers about 150'000 terms but it lacks of information about the relation of these

terms. For example, the relation between the terms Beijing and Beijing Olympics could not been explained by CIS.

To fix this and other lacks, Kobilarov et al. (2009) decided to search for a common set of web identifiers. They decided to use DBpedia identifiers, given the popularity of this project and its central role in the LOD cloud. Consequently, the vocabulary of DBpedia was used to connect the BBC domains. To match the CIS terms with the DBpedia identifiers an algorithm was applied. Kobilarov et al. (2009) have announced to expand the successfully applied proceeding of the domain programmes to other BBC domains to increase the connectivity of the microsites and thus to increase the potential of the BBC online resources.

2.3 Streaming Linked Data

This chapter covers related work done in the field of streaming Linked Data. First of all, we present a proposal for publishing data streams as Linked Data. This is followed by two projects done at the University of Ghent in Belgium. At the end of this chapter we introduce the framework TripleWave which represents an important part of this Bachelor Thesis.

Barbieri & Della Valle (2010) mention that they are working on a query language, which can evaluate continuous streams of RDF and static RDF graphs. The query language is called C-SPARQL. Barbieri & Della Valle (2010) present in their position paper an extension of their C-SPARQL Engine. The C-SPARQL Engine can stream Linked Data. The purpose of the extension is to make it easier for Web applications to work with data streams.

The C-SPARQL Engine consumes data streams, RDF streams and RDF Graphs. Then a client can register a continuous query, which evaluates over the incoming streams. It is even possible to remotely register a query. Barbieri & Della Valle (2010) implemented a RESTful interface, which enables to register new queries, start and stop them & to delete queries. The query results are published as Linked Data by a special local C-SPARQL client named Streaming Linked Data Server. Thus the results can be consumed by Linked Data clients (Barbieri & Della Valle 2010).

Barbieri & Della Valle (2010) also present a concept of how an RDF stream should be built. They mention that an RDF stream consists of an ordered sequence of pairs. Every pair includes an RDF triple and a time stamp. An RDF stream should be represented as named graphs. There are two sorts of named graphs: the s-graphs (Stream Graphs) and the i-graphs (Instantaneous Graphs). An s-graph contains the meta data about the current window of the stream. An i-graph represents the particular triples which contain the same time stamp. In the proposal, they suggest how to build a named graph with the RDF format. Moreover, they introduce a concept of how clients can define the different sorts of windows (logical windows and physical windows) which should be used in the C-SPARQL query.

Barbieri & Della Valle (2010) note that the presented concepts better match the characteristics of streams as other studies done in the field of streaming Linked Data. In particular the introduced concepts of the windows and the representation of the named graphs as RDF suit the fact that streaming data are not persistent but transient.

Heyvaert et al. (2016) present in their paper an approach of how linked sensor data can be generated by using the RDF Mapping Language (RML) and a Triple Pattern Fragments (TPF) server. A special characteristic of their approach is that it can be used for several data sources and the mapping is easily reusable and modifiable. Heyvaert et al. (2016) describe the prototype as follows: An application consumes the sensor data created by a Tessel module. RML mappings are stored on the TPF server, which can be fetched by the application. Then the application uses the RML Mapper to apply the corresponding RML mapping on the sensor data. As a result, the application receives the linked sensor data, which can be forwarded to RDF-based applications.

Heyvaert et al. (2016) mention two challenges which their prototype can meet. The first challenge is to find and use an RDF transformation which can handle different data sources in different data formats. As sensor data normally originate from different sources with different data formats, it is important to

create a prototype which can handle this in a convenient way. RML makes that possible by "offering a declarative way to define how data in multiple heterogeneous data sources is mapped to RDF triples" (Heyvaert et al. 2016, p. 2). The second challenge they name is that requirements for applications can change fast. To meet this challenge, the prototype must offer an easy way of how mappings for the sensor streams can be changed. This is achieved by storing the RML mappings on the TPF server. This means, that if changes are required, the user only must access the server and change the mapping. No changes on the application itself are required due to the modularity of the prototype.

Taelman et al. (2016a) present an approach which builds on the above one. They describe an approach of how sensor data can be consumed, converted to RDF, published and finally evaluated by a continuous query. The sensor data are again generated by a Tessel sensor. The data can be easily readout because Tessel offers a Node.js module for their sensors. This module returns the sensor data in the JSON format. The transformation of the sensor data into linked sensor data as RDF is performed by the prototype presented by Heyvaert et al. (2016).

In the third step, publishing the data, a time annotation is added to the linked sensor data. Thus the data can be chronologically ordered. Taelman et al. (2016a) have decided to represent the data as RDF graph because they have shown in a former paper (Taelman et al. 2016b) that this is the most efficient approach. A TPF server publishes these dynamic linked sensor data. One advantage of such a TPF server is that a client application can easily set up a TPF Query Streamer. A TPF Query Streamer can evaluate the data provided by a TPF server using a continuous query. Taelman et al. (2016a) also mention that using a TPF server enables to store old linked sensor data, so that the history of the linked sensor data can be analysed.

Mauri et al. (2016) present the framework TripleWave, which is used in this Bachelor Thesis and therefore plays an important role in it. TripleWave closes a gap by offering a flexible and generic solution which is able to publish Linked Data streams on the Web. It was important for Mauri et al. (2016) that their framework follows the Semantic Web standards, that the output data can be accessed in different ways and that it is able to consider various data source configurations.

The requirements for TripleWave are influenced by real-world use cases. TripleWave can run in three different modes. One mode allows TripleWave to consume a non-RDF stream which is then transformed into an RDF stream. For example, the Wikimedia live stream, which shows the recent changes and is available in JSON, can be taken as input. TripleWave then uses an easily customizable R2RML mapping to transform the non-RDF input stream to an RDF stream serialized in the JSON-LD format[17]. The second mode enables to stream out a static RDF data set which includes time annotations. The time annotations are necessary to order the RDF data chronologically. Once the whole RDF data set is streamed out, the execution of TripleWave has finished. The third mode streams out a static RDF data set with time annotations in an endless loop. To do so, the time stamps of the RDF data are continually increased and thus TripleWave can simulate an endless stream. This is especially useful for benchmarking or testing applications.

Mauri et al. (2016) mention that the output stream of TripleWave is available as JSON-LD and follows the Linked Data Principles. TripleWave follows the proposal of Barbieri & Della Valle (2010) mentioned above. This means that the RDF stream consists of a Stream Graph (s-Graph) and Instantaneous Graphs (i-Graphs). The s-Graph can be accessed through an URL. The i-Graphs are the content of the actual stream. The output stream of TripleWave is available in a push-based and a pull-based approach. A push-based connection is established by using WebSockets. All these characteristics show that TripleWave is a flexible framework and therefore it can be used in various scenarios.

[17] https://json-ld.org/ (accessed 11.7.2017)

2.4 Kafka and Thrift

The approaches we mention in Section 2.3 "Streaming Linked Data" are prototypes developed in research projects. There are more solid frameworks which are used by companies for production services. This section introduces Apache Kafka and Apache Thrift. Kafka is a well-known framework for processing huge amounts of data in near real-time. We use Kafka to coordinate a cluster of several OGD streams and thus is another important resource for this thesis. We use Apache Thrift as a JSON-LD alternative to serialize RDF streams.

Kreps, Narkhede & Rao (2011) present Kafka, a framework used at LinkedIn for log processing. Kafka was developed because the available messaging systems did not meet the requirements which LinkedIn needed for real-time log processing. Applications like Cloudera's Flume, Yahoo's Data Highway or Facebook's Scribe were developed for collecting the log data from the client applications and then forward them into Hadoop or data warehouses for offline evaluation. But LinkedIn wanted an application which is able to not only make these huge amounts of log data available for offline consumption, but also can provide these data in near real time to different applications. The result of implementing these requirements is Kafka, a distributed and highly scalable framework for processing huge amounts of data.

Kreps, Narkhede & Rao (2011) express the Design Principle of Kafka as follows: streams of related messages which pass Kafka are defined by topics. Producers are able to send messages to the Kafka cluster associating them to certain topics. The Kafka cluster consists of servers (called brokers) which store these messages. A partition is the smallest entity of parallelism in Kafka. A topic can be divided into several partitions which then are distributed among the brokers. This enables to process data about a certain topic in parallel over several brokers. A consumer or a consumer group is able to consume data from the Kafka cluster through a pull mechanism by subscribing to the desired topic. A consumer group consists of several consumers, which altogether consume the data of a certain topic. Design decisions were made in a way such that messages require as few overhead as possible and that the Kafka cluster has to do as few coordination as possible to achieve high throughput. For example, it was decided that the brokers are stateless, meaning that the brokers do not coordinate the consumer or the consumer groups. There is no central master node implemented. Instead, Kafka uses Zookeeper[18] for coordinating the components. Also the storage and the transfer of the data are kept as easy and efficient as possible through certain design decisions.

Kreps, Narkhede & Rao (2011) decided that Kafka does not guarantee exactly-once delivery, as this would require a two-phase commit. This would lead to more complexity which would decrease the throughput. Kafka instead guarantees at-least-once delivery. Usually the data are delivered exactly-once. Only if consumers crash there is the possibility that some messages are delivered twice. At the time Kreps, Narkhede & Rao (2011) published their paper, LinkedIn processed every day hundreds of gigabytes of log data. The advantage of Kafka providing high scalability and high throughput outweighs the fact that there is the chance some data can be duplicated.

Wang et al. (2015) present in their short paper the extension of Kafka into a replicated logging system. Before this extension the data in Kafka were not replicated. If a Kafka broker crashed and its storage was damaged, all the data which was not consumed was lost. To minimize this risk, Kafka was extended into a replicated logging system.

To do so Wang et al. (2015) mention two needed requirements. First, a protocol is necessary which keeps the replicated logs on the different severs consistent. That means, the replicated logs must have the same content in the same order among the distributed servers. It was decided to implement message log replication "with the key idea of separating the key elements of a consensus protocol such as leader election and membership changes from log replication itself" (Wang et al. 2015, p. 1654). With such an approach the log format is less complex and more flexibility is achieved. The second requirement names a mechanism to truncate the replicated logs to prevent them from growing endlessly. Until this extension

[18] https://zookeeper.apache.org/ (accessed 11.7.2017)

Kafka included only a truncation mechanism which based on time or space windows. These approaches are not sufficient for key-based logs, since the system may need only the newest data. Therefore, in Kafka 0.8.1 an additional mechanism was introduced which handles the truncation of replicated key-based log entries.

Vineet & Xia (2017) compare in their paper the performance of the two distributed message brokers, Kafka and RabbitMQ. One big difference between Kafka and RabbitMQ shall be remarked at this point. In Kafka the design principles prefer throughput over reliability. In RabbitMQ, it is the opposite case. RabbitMQ uses the AMPQ protocol which was designed for usage in the financial and banking sector. Thus reliability takes priority in RabbitMQ because loosing financial transactions should not happen.

Vineet & Xia (2017) built a testbed which contained five similar nodes. They performed two different kinds of benchmark tests. The first measured the performance of a single producer/consumer pair when the number of nodes changes (from one to five). The workload in this test was constant. The other test measured the performance using a constant number of nodes and changing the number of producer and consumers. In each case the throughput and latency were measured. The results show that Kafka can handle more throughput than RabbitMQ but RabbitMQ has generally the lower latency. The higher throughput of Kafka is explained with certain design decisions made in Kafka like using the SendFile API, the sequential disk writing and the out-of-the-box batching. All these characteristics support higher throughput. RabbitMQ can achieve the lower latency because in the default configuration no disk writing is executed and the connection between the broker and consumer is established by a push model.

Vineet & Xia (2017) conclude that the characteristics throughput and reliability are crucial when choosing one of these two systems. If an at-least-once delivery is sufficient, then they recommend choosing Kafka. But if reliability is more important than throughput, then one should choose an application which uses the AMPQ protocol like RabbitMQ.

Slee, Agarwal & Kwiatkowski (2007) introduce in their paper the framework Apache Thrift, which was developed at Facebook. Thrift is a serialization, deserialization and RPC framework. Facebook has a lot of services implemented in different programming languages. This created the challenge of implementing high-performance connections between different languages. Because no satisfying framework was available, Facebook decided to develop its own.

Slee, Agarwal & Kwiatkowski (2007) explain that in a thrift file one can define attributes, objects and interfaces for functions. There are some basic types available which can be used to define attributes. These types are usually available in any language. There are also lists, sets and maps available which are called containers. Using these containers and base types, structs can be defined, which are like objects in object-oriented languages. The interfaces for functions are called services. Thrift also allows creating exceptions. When a developer has defined the desired structs and services, the Thrift compiler can be used to generate files in various languages which implement the defined structs and contain stubs for the interfaces. Thrift supports now over 15 different programming languages[19]. Thrift handles the serialization and deserialization of the data into the desired languages and also manages the transport between the applications. An application developer only has to write the thrift file with the struct and service definition, compile that file and implement the generated interfaces. All the rest is handled by Thrift.

Slee, Agarwal & Kwiatkowski (2007) described that Thrift is used in several Facebook services like Facebook search and logging, because the backend systems are written in languages like C++ and Python but the Frontend is written in PHP. Thrift allows the developer to focus on writing the applications ignoring the challenge of connecting applications written in different languages. Other big companies like Cloudera, Evernote, Siemens or Uber use Thrift in their production services[20].

[19] https://thrift.apache.org/lib/ (accessed 14.6.2017)

[20] https://thrift.apache.org/about (accessed 14.6.2017)

3 A survey of streaming data in German-language Open Government Data

This chapter gives a survey of available streaming data in Swiss Open Government Data with some extra data sets from German and Austrian OGD portals. The chapter is built as follows: in Section 3.1 there is a short introduction of the 5-star open data scheme proposed by Tim Berners-Lee. Section 3.2 describes our approach of how the data sets were found which are presented and analysed afterwards. The result of this chapter is a survey with the identified data sets qualified as possible streaming data.

3.1 5-Star open data

In 2010, Tim Berners-Lee, who is known as the inventor of the Web, developed a 5-star open data scheme. The higher an open data set is rated corresponding to this scheme, the more valuable the open data set becomes. The different levels build on each other. This means that 2 stars' open data must fulfil the criteria for 1-star open data plus the additional criteria for the second star. The following description of this 5-star open data rating system bases on Tim Berners-Lee's rating system[21] and we add examples for better understanding:

1 star: the idea of the first level of this open data rating system is that the data provider publishes his data **under an open license**. The format is not important; the main point is that the data are published under an open license and thus can be used by any other user. As an example, let's consider the data set published at: *https://www.govdata.de/web/guest/suchen/-/details/statistisches-jahrbuch-2009-ulm* (accessed 16.5.2017). This data set is available under an open license (CC-BY) and in the format of some PDFs. Therefore, it can be used by anyone but it is not machine readable.

2 stars: the open data set gets a second star if it is published in a **machine-readable way**. For example, if the data provider publishes tabular data in the format of an Excel file instead of an image scan of the table, the document is a candidate for the second star. The data set published at the following link is an example for 2 stars' open data: *https://data.stadt-zuerich.ch/dataset/wir-preise-zik-basis-dez2010* (accessed 16.5.2017).

3 stars: the meaning of this level is that the data are available **in a non-proprietary format**, so that the data use is not depending on owning a certain program to read the data. This is the case if data are for example published in a txt format instead of using a MS Word file. A good example for a 3 stars' open data set is following link: *https://data.stadt-zuerich.ch/dataset/vbz_fahrplandaten_gtfs* (accessed 16.5.2017).

4 stars: to raise open data to a 4 stars' level, the data set must use **URIs to label the entities** which are mentioned in the data set. There are open standards from the W3C (e.g. RDF) which describe how this requirement can be implemented. At the best of our knowledge no German-language Open Government Data files have this level of open data.

5 stars: this is the highest level of open data and provides the biggest benefits for the open data users. To reach this level, the provider must **link the data to other data to create context**. If this is done, the machine which uses the data has the possibility to access other data sets with related information which are also linked to other data sets and is therefore able to produce results with higher quality. An example is available at: *https://opendata.swiss/de/dataset/bibliografische-daten-rdf* (accessed 16.5.2017).

The following image illustrates these five different levels of open data:

[21] https://www.w3.org/DesignIssues/LinkedData.html (accessed 16.5.2017)

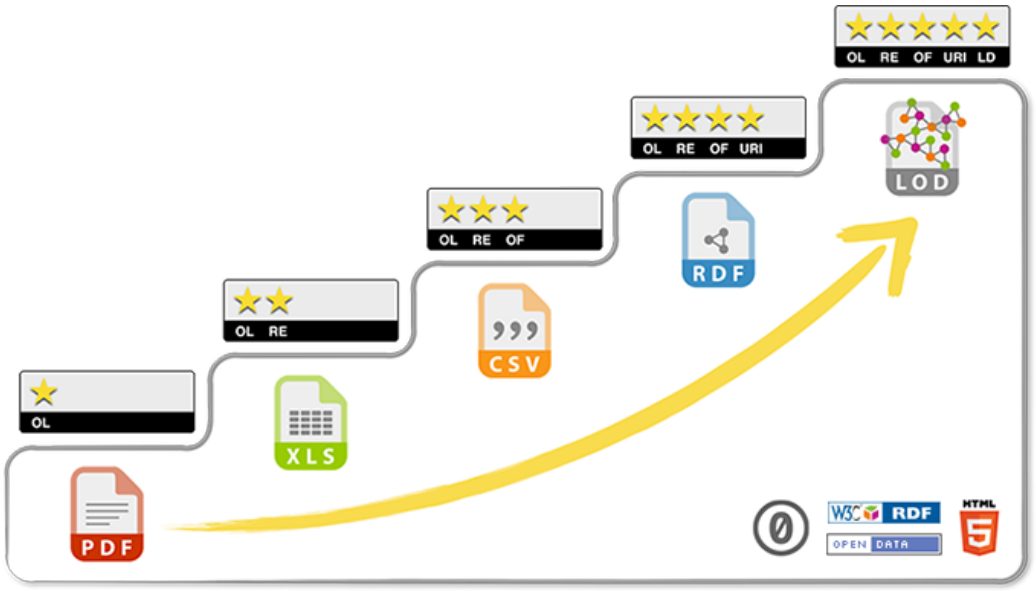

Figure 1: 5-star open data rating system illustrated as an image, source: (Hausenblas, 2015)

3.2 Available streaming data in German-language Open Government Data

To select specific real-time streams or data sets to be streamed out to the Web, we must first analyse the available OGD and then build a survey. The scope of the analysis is limited to the Swiss OGD with some extra search performed in the German and Austrian OGD portals. In this section the results of analysing the German-language OGD are presented. Section 3.2.1 describes how and which OGD portals were examined. Section 3.2.2 shows and explains the survey table which is the result of the examination. In Section 3.2.3 we analyse the survey.

3.2.1 Approach of searching thoroughly the Open Government Data portals

This section describes how we examined the Open Government Data portals. As a result of this examination, we build a survey table which contains data sets that are characterized by high dynamicity. This survey table is a valuable resource for further research in the areas of OGD and Stream Processing. To not go beyond the scope of this Bachelor Thesis the searching for potential streaming data is mainly restricted to the Swiss OGD portals with some additional search performed in the German and Austrian OGD portals.

In the first step we examined the following two sites:

- *https://data.stadt-zuerich.ch/* (accessed 15.5.2017)
- *https://opendata.swiss/de/* (accessed 15.5.2017)

Data sets which are especially suitable for getting streamed out to the Web contain real-time data. To find such data sets, we searched for the following keywords:

- real-time
- real time
- *Echtzeit* (real time)
- *echtzeit* (real time)
- live

We manually checked the data sets that we found, if they are suitable for getting used as streaming data. That means, the data sets should change or be updated frequently. The resulting data sets are the following ones (the characters in the brackets after the titles represent the short names of the data sets which can be looked up in Table 1; after the brackets we give an English translation of the titles):

- *Parkleitsystem: Echtzeitinformationen zu freien Parklplätzen in verschiedenen Parkhäusern* (CPZH) – Parking guidance system: real-time data about free parking spaces in different parking facilities
- *Verfügbarkeit der Velos von "Züri rollt" (real time)* (ZR) – Availability of bicycles from "Züri rollt" (real time)
- *Wassertemperatur Freibäder* (WTZH) – Water temperature of open air swimming pools
- *Stündlich aktualisierte Luftqualitätsmessungen* (HAQMZH) – Hourly updated air quality measurements
- *Echtzeitdaten am Abstimmungstag* (VDZH) – Real-time data at voting days

Since we found only five data sets, we manually examined the two OGD sites. For the OGD portal of Zurich we examined all data sets. For the OGD portal of Switzerland we did not examine all the data sets because of the high number of available data sets. The focus of this manual examination was on data sets which are updated frequently and contain records with time stamps. This is because data sets which are not updated at all or are updated only rarely are not interesting for getting streamed out. Such data sets rarely deliver new data if at all.

The time stamp is important to order the records in the data set. It came out that particularly data sets from the categories mobility and environment fulfil these two criteria. We have found the following data sets:

- *Monatlich aktualisierte Luftqualitätsmessungen (seit 2012)* (MAQMZH) – Monthly updated air quality measurements (since 2012)
- *Täglich aktualisierte Luftqualitätsmessungen* (TAQMZH) – Daily updated air quality measurements
- *Daten der automatischen Fussgänger- und Velozählung – Viertelstundenwerte* (TCZH) – Data of automatic pedestrians and cyclists count – quarter-hourly values
- *Fahrzeiten der VBZ im Soll-Ist-Vergleich* (PTZH) – Target-actual comparison of VBZ's travel times
- Messwerte der Wetterstationen der Wasserschutzpolizei Zürich (WSZH) – Weather measurements of Zurich Water Police
- «Züri wie neu» - Meldungen (ZWNZH) – «Züri wie neu» - reports
- *Messdaten SMN Niederschlag* (SMNRCH) – Measurement data SMN rainfall
- *Messdaten SMN* (SMNCH) – Measurement data SMN

To find more suitable data sets we spread the searching to the following two public transportation sites:

- *https://opentransportdata.swiss/en/* (accessed 15.5.2017)
- *https://data.sbb.ch/explore/?sort=modified* (accessed 15.5.2017)

The number of published data sets on these two portals is not too big so we manually examined them and without keywords. The focus was again on data sets which are updated frequently and which contain time stamps. We found the following data sets:

- *Trip forecast* (TFCH)
- *Departure / arrival display* (DADCH)
- *Actual Data* (ADSBB)

- *Actual Data – History* (ADHSBB)
- *Actual Data - Previous Day* (ADPDSBB)

To find further Swiss data sets we looked for other OGD portals. The goal was to find portals which are maintained by German-language cities in Switzerland. If Zurich maintains such a portal, then we hypothesized that other cities do it as well. We searched for portals of German-language cities with more than 30'000 inhabitants according to Wikipedia[22] and for portals of the main cities of the German-language Cantons of Switzerland. But we did not find additional useful OGD portals.

We then spread the examination for suitable data sets to some OGD sites of Germany and Austria. These portals can be viewed by calling the following links:

- *https://www.govdata.de/* (accessed 15.5.2017)
- *https://www.data.gv.at/* (accessed 15.5.2017)

We repeated the search by the following keywords:

- real-time
- real time
- *Echtzeit (real time)*
- *Echtzeit (real time)*
- live

We have found 22 data sets and we manually checked them. The resulting data sets are the following ones:

- *Stadt Moers: Parkleitsystem Moers* (CPMDE) – City of Moers: Parking guidance system Moers
- *Stadt Bonn: Parkhäuser (Parkhausbelegung)* (CPBDE) – City of Bonn: Car parks (car park occupancy)
- *Stadt Bonn: aktuelle Strassenverkehrslage* (CTBDE) – City of Bonn: current road situation
- *Lufttemperatur T* (ATDE) – Air temperature T
- *Wiener Linien – Echtzeitdaten* (WPTAU) – Vienna's public transportation routes – real-time data
- *Parkplätze in der Stadt Salzburg* (CPSAU) – Car parks in the city of Salzburg

The survey table shows after this step a reasonable number of potential streaming data sets. So we did not do further manual examination. We did also not search for further OGD portals of German or Austrian cities. This would go beyond the scope of this Bachelor Thesis. As written above not all OGD portals were fully examined thus the survey table (see Tables 1-6) is non-exhaustive. It is a solid starting point for further research and can or should be extended if needed.

3.2.2 Description of the survey table

This section presents the results of our survey. A tabular representation is available in Tables 1-6. We describe the columns of the tables and give comments to certain data sets if necessary.

The first column is called Title. It contains the title of the found data sets. If the title is used to search for the corresponding data set on the corresponding portal, then the user will be redirected to this data set.

[22] https://de.wikipedia.org/wiki/Liste_der_St%C3%A4dte_in_der_Schweiz (accessed 15.5.2017)

Short name	Title
Swiss Open Government Data:	
CPZH	Parkleitsystem: Echtzeitinformationen zu freien Parklplätzen in verschiedenen Parkhäusern
ZR	Verfügbarkeit der Velos von "Züri rollt" (real time)
WTZH	Wassertemperatur Freibäder
MAQMZH	Monatlich aktualisierte Luftqualitätsmessungen (seit 2012)
TAQMZH	Täglich aktualisierte Luftqualitätsmessungen
HAQMZH	Stündlich aktualisierte Luftqualitätsmessungen
TCZH	Daten der automatischen Fussgänger- und Velozählung - Viertelstundenwerte
PTZH	Fahrzeiten der VBZ im Soll-Ist-Vergleich
WSZH	Messwerte der Wetterstationen der Wasserschutzpolizei Zürich
ZWNZH	«Züri wie neu» - Meldungen
VDZH	Echtzeitdaten am Abstimmungstag
SMNRCH	Messdaten SMN Niederschlag
SMNCH	Messdaten SMN
TFCH	Trip forecast
DADCH	Departure / arrival display
ADSBB	Actual Data
ADHSBB	Actual Data - History
ADPDSBB	Actual Data - Previous Day
German/Austrian Open Government Data:	
CPMDE	Stadt Moers: Parkleitsystem Moers
CPBDE	Stadt Bonn: Parkhäuser (Parkhausbelegung)
CTBDE	Stadt Bonn: aktuelle Strassenverkehrslage
ATDE	Lufttemperatur T
WPTAU	Wiener Linien - Echtzeitdaten
CPSAU	Parkplätze in der Stadt Salzburg
Outlook:	
GTFSR	GTFS Realtime

Table 1: Survey Table Part 1

Short name	Provider	License
Swiss OGD:		
CPZH	Open Data Zürich	Creative Commons CCZero
ZR	Open Data Zürich	Creative Commons CCZero
WTZH	Open Data Zürich	Creative Commons CCZero
MAQMZH	Open Data Zürich	Creative Commons CCZero
TAQMZH	Open Data Zürich	Creative Commons CCZero
HAQMZH	Open Data Zürich	Creative Commons CCZero
TCZH	Open Data Zürich	Creative Commons CCZero
PTZH	Open Data Zürich	Creative Commons CCZero
WSZH	Open Data Zürich	Creative Commons CCZero
ZWNZH	Open Data Zürich	Creative Commons CCZero
VDZH	Statistisches Amt Kanton Zürich	https://opendata.swiss/en/terms-of-use/
SMNRCH	Bundesamt für Meteorologie und Klimatologie MeteoSchweiz	https://opendata.swiss/en/terms-of-use/
SMNCH	Bundesamt für Meteorologie und Klimatologie MeteoSchweiz	https://opendata.swiss/en/terms-of-use/
TFCH	opentransportdata.swiss	https://opentransportdata.swiss/en/terms-of-use/
DADCH	opentransportdata.swiss	https://opentransportdata.swiss/en/terms-of-use/
ADSBB	opentransportdata.swiss	https://opentransportdata.swiss/en/terms-of-use/
ADHSBB	SBB	https://data.sbb.ch/page/licence/
ADPDSBB	SBB	https://data.sbb.ch/page/licence/
DE/AU OGD:		
CPMDE	Stadt Moers	CC BY-NC-SA 4.0
CPBDE	Stadt Bonn	cc-by-nc
CTBDE	Stadt Bonn	Creative Commons Namensnennung (CC-BY)
ATDE	Landesamt für Natur, Umwelt und Verbraucherschutz NRW	Datenlizenz Deutschland Namensnennung 2.0
WPTAU	Stadt Wien	Creative Commons Namensnennung 3.0 Österreich
CPSAU	Stadt Salzburg	Creative Commons Namensnennung 3.0 Österreich
Outlook:		
GTFSR	opentransportdata.swiss	https://opentransportdata.swiss/en/terms-of-use/

Table 2: Survey Table Part 2

Short name	Link
Swiss OGD:	
CPZH	https://data.stadt-zuerich.ch/dataset/parkleitsystem
ZR	https://data.stadt-zuerich.ch/dataset/mietvelo-verfuegbarkeit
WTZH	https://data.stadt-zuerich.ch/dataset/wassertemperaturen-freibaeder
MAQMZH	https://data.stadt-zuerich.ch/dataset/luftqualitaet-historisierte-messungen
TAQMZH	https://data.stadt-zuerich.ch/dataset/luftqualitaet-tages-aktuelle-messungen
HAQMZH	https://data.stadt-zuerich.ch/dataset/luftqualitaet-stunden-aktuelle-messungen
TCZH	https://data.stadt-zuerich.ch/dataset/verkehrszaehlungen-werte-fussgaenger-velo
PTZH	https://data.stadt-zuerich.ch/dataset/vbz-fahrzeiten-ogd
WSZH	https://data.stadt-zuerich.ch/dataset/sid_wapo_wetterstationen
ZWNZH	https://data.stadt-zuerich.ch/dataset/zueriwieneu-meldungen
VDZH	https://opendata.swiss/de/dataset/echtzeitdaten-am-abstimmungstag
SMNRCH	https://opendata.swiss/de/dataset/messdaten-smn-niederschlag-swissmetnet
SMNCH	https://opendata.swiss/de/dataset/messdaten-smn-swissmetnet
TFCH	https://opentransportdata.swiss/en/dataset/fahrtprognose
DADCH	https://opentransportdata.swiss/en/dataset/aaa
ADSBB	https://opentransportdata.swiss/en/dataset/istdaten
ADHSBB	https://data.sbb.ch/explore/dataset/ist-daten-history/
ADPDSBB	https://data.sbb.ch/explore/dataset/actual-data-sbb-previous-day/
DE/AU OGD:	
CPMDE	https://www.govdata.de/web/guest/suchen/-/details/parkleitsystem-moers-odp
CPBDE	https://www.govdata.de/web/guest/suchen/-/details/parkhaeuser-bn
CTBDE	https://www.govdata.de/web/guest/suchen/-/details/aktuelle-strassenverkehrslage-innenstadt-bn
ATDE	https://www.govdata.de/web/guest/suchen/-/details/kontiluqs-t
WPTAU	https://www.data.gv.at/katalog/dataset/add66f20-d033-4eee-b9a0-47019828e698
CPSAU	https://www.data.gv.at/katalog/dataset/9087fe9a-1dd4-49a1-98b4-8a8c659eb64f
Outlook:	
GTFSR	https://opentransportdata.swiss/en/dataset/gtfsrt

Table 3: Survey Table Part 3

Short name	Real-time	Publication Frequency of Data Set	Update Frequency of Data Set	Precision of Time Stamp
Swiss OGD:				
CPZH	yes	continuous	continuous	second
ZR	yes	continuous	continuous	no
WTZH	no	at least 2 times a day	at least 2 times a day	minute
MAQMZH	no	monthly	daily	day
TAQMZH	no	daily	daily	day
HAQMZH	no	hourly	hourly	minute
TCZH	no	weekly	every 15 minutes	second
PTZH	no	weekly	n/A	second
WSZH	no	every 10 minutes	every 10 minutes	second
ZWNZH	no	weekly	n/A	day
VDZH	yes	continuous	continuous	no
SMNRCH	no	every 10 minutes	every 10 minutes	minute
SMNCH	no	every 10 minutes	every 10 minutes	minute
TFCH	yes	continuous	continuous	second
DADCH	yes	continuous	continuous	second
ADSBB	no	daily	n/A	second
ADHSBB	no	daily	n/A	second
ADPDSBB	no	daily	n/A	second
DE/AU OGD:				
CPMDE	yes	continuous	continuous	no
CPBDE	yes	continuous	continuous	minute
CTBDE	no	every 5 minutes	every 5 minutes	second
ATDE	no	daily	hourly	second
WPTAU	yes	continuous	continuous	no
CPSAU	no	every 5 minutes	every 5 minutes	minute
Outlook:				
GTFSR	yes	continuous	continuous	second

Table 4: Survey Table Part 4

Short name	Number of Records	Representation	Number of Stars
Swiss OGD:			
CPZH	37	XML RSS-Feed	3
ZR	8	XML	3
WTZH	16	XML	3
MAQMZH	2k	CSV	3
TAQMZH	30	CSV	3
HAQMZH	169	CSV	3
TCZH	1-2M per year; since 2009	CSV	3
PTZH	1-2M per week; since 2015	CSV	3
WSZH	n/A	JSON	3
ZWNZH	10k	GeoJson, KMZ, WMS, WFS, GPKG, XML	3
VDZH	183	JSON	3
SMNRCH	96	CSV	3
SMNCH	113	CSV	3
TFCH	n/A	XML	3
DADCH	n/A	XML	3
ADSBB	100k-1M	CSV	3
ADHSBB	1-10M	CSV, JSON, Excel, GeoJson, KML	3
ADPDSBB	10-100k	CSV, JSON, Excel, GeoJson, KML	3
DE/AU OGD:			
CPMDE	13	XML	3
CPBDE	6	XML	3
CTBDE	93	GeoJson	3
ATDE	9k	CSV	3
WPTAU	n/A	JSON	3
CPSAU	30	GML, JSON, CSV, ESRI Shapefile, KML, GeoRSS	3
Outlook:			
GTFSR	n/A	JSON, XML	3

Table 5: Survey Table Part 5

Short name	Comment
Swiss OGD:	
CPZH	
ZR	
WTZH	Only updated in the summer
MAQMZH	
TAQMZH	
HAQMZH	
TCZH	
PTZH	
WSZH	API: no registration necessary
ZWNZH	There is an Open311 API available
VDZH	Only updated at election day
SMNRCH	Download link for data set: http://data.geo.admin.ch/ch.meteoschweiz.swissmetnet-niederschlag/VQHA70.csv
SMNCH	Download link for data set: http://data.geo.admin.ch/ch.meteoschweiz.swissmetnet/VQHA69.csv
TFCH	API: registration necessary
DADCH	API: registration necessary
ADSBB	
ADHSBB	Data set can be exported or accessed by API
ADPDSBB	Data set can be exported or accessed by API
DE/AU OGD:	
CPMDE	
CPBDE	
CTBDE	
ATDE	Temperature of Nordrhein-Westfalen
WPTAU	API: registration necessary
CPSAU	Only 3 of the 30 records are certainly updated every 5 minutes. Poor Quality
Outlook:	
GTFSR	Beta version published on 22.5.2017. Not all services are already available. Only two calls per minute allowed

Table 6: Survey Table Part 6

The column *Provider* shows the organisation that provides the data set. Column *License* contains the license or the terms of use under which a data set is published. If the license has a specific name, then this name is represented e.g. Creative Commons CCZero. In other cases, the link to the license or to the terms of use is represented. The OGD portal describes the duties in words which come along with the usage of their data sets and must be looked up under the given link.

The column *Link* contains the URL that redirects the user to the Web site where the data set is hosted. If the data set is available on more than one portal, the link is given to that portal, which is more specific. For example, the data set *Parkleitsystem: Echtzeitinformationen zu freien Parklplätzen in verschiedenen Parkhäusern* is available under the link *https://data.stadt-zuerich.ch/dataset/parkleitsystem* as well as under the link *https://opendata.swiss/de/dataset/parkleitsystem-echtzeitinformation-zu-freien-parkplatzen-in-verschiedenen-parkhausern*. Since the city of Zurich is the provider of this data set, the column contains the URL to the data set on the OGD portal of Zurich.

The column *Real-time* describes if the data set contains real-time data according to the provider. If this is the case, then the columns *Publication Frequency of Data Set* and *Update Frequency of Data Set* contain the value *continuous*.

The column *Publication Frequency of Data Set* describes in which time interval a data set is published. For example, if there is written *hourly*, then the data set is normally published once an hour. The next column, named *Update Frequency of Data Set*, describes how often the data set is updated by the provider. Not all providers publish their data set after each update. For example, the data set *Daten der automatischen Fussgänger- und Velozählung – Viertelstundenwerte* contains records for every 15 minutes (which means that updates respectively new records are added every 15 minutes), but the data set is published only once a week.

The column *Precision of Time Stamp* relates to the records' time stamps, and contains the smallest unit of the time stamp. For example, the term *second* means that the time stamp contains seconds as smallest unit. If a data set's records do not have time stamps, then there is written *no*.

The column *Number of Records* describes the size of a data set. If a data set contains information about objects, then the number of objects is listed in this column. For example, the data set *Parkleitsystem: Echtzeitinformationen zu freien Parklplätzen in verschiedenen Parkhäusern* contains information about parking spots in Zurich. So the column contains the number of parking spots. Such data sets contain a small number of entities (<1000) and the number does normally not change. This is why we have written down the exact number of entities respectively objects. The other data sets are much bigger and contain more than 1000 records. Often the number of records changes over time and therefore only the order of magnitude of the number of records is given.

The column *Representation* describes in which format a data set is available. If there are more than one format, all of them are reported.

The column *Number of Stars* contains the level of open data as described in Section 3.1.

The last column, *Comments*, contains additional notes if necessary. Following, we explain the comments more detailed:

- *Wassertemperatur Freibäder* (WTZH): This data set is only updated in summer, because it describes the water temperature of the outdoor pools in the city of Zurich. So a seasonal streaming would be interesting whereas streaming the data the whole year would not make a lot of sense.
- *Echtzeitdaten am Abstimmungstag* (VDZH): This data set includes the results at voting days. It is updated only in these days. Maintaining a stream at the voting days would increase the usability of those data, but for this Bachelor Thesis it is not of big use, because at most of the time no evaluation can be made.
- *«Züri wie neu» - Meldungen* (ZWNZH): This data set can be downloaded in several formats or can be accessed by API. The API uses the Open311 standard.

- *Messdaten SMN Niederschlag* (SMNRCH) & *Messdaten SMN* (SMNCH): The comment contains the direct downloading link for the corresponding data set. This is because it is tricky to find these download links. If one wants to download the data set, the link on the *https://opendata.swiss* platform redirects the user to *http://data.geo.admin.ch*. There one has to find the relating link and to download a package which includes a ReadMe file. In this ReadMe file the download link is available.
- *Trip forecast* (TFCH) & *Departure / arrival display* (DAPCH): The data of these two data sets can only be accessed by API calls. The user has to register to be able to access this API.
- *Actual Data – History* (ADHSBB) & *Actual Data - Previous Day* (ADPDSBB): These two data sets are offered by SBB (*Schweizerische Bundesbahnen*), which runs the biggest part of the rail network in Switzerland. It offers data sets in two ways: As data dumps to be downloaded, or as data accessible through API calls. The latter makes sense if only parts of the whole data sets are needed because the whole data sets contain quite a big number of records.
- *Lufttemperatur T* (ATDE): This data set contains the temperature measurements of the state Nordrhein-Westfalen. This is noted, because at first glance it is not clear, from where these temperature data originate.
- *Wiener Linien – Echtzeitdaten* (WPTAU): This data set can only be accessed through API calls. To be able to make those calls, the user has to fill out a registration form and has to mention the reason why he/she wants access. In our case, the registration form was processed within 24 hours.
- *Parkplätze in der Stadt Salzburg* (CPSAU): This data set suffers from poor quality. Out of 30 records, only three were real-time updated. But this data set is mentioned on the list, because maybe the quality increases in future, so the benefit of using it will also increase.

There is one data set listed at the very bottom of the survey table. The *https://opentransportdata.swiss* portal has released a beta version of this API on 22nd Mai 2017. There are not all services available yet. A registration is needed to get access to this API. The API provides GTFS real-time data[23]. It is possible to submit two calls per minute. This real-time feed contains all known changes for the public transportation of Switzerland including a time window of three hours for all public transportation companies which provide real-time data.

3.2.3 Analysis of the survey table

We analyse in this section the survey illustrated in Tables 1-6. As mentioned above it is possible that we did not find all data sets which are suitable for being streamed out to the Web. So the analysis done in this chapter is not representative for the entirety of potential open streaming data sets available in the German-language open data portals.

The table contains 25 data sets. 19 data sets originate from Swiss open data portals, four data sets are German ones and two data sets were found on the Austrian open data portal.

Figure 2 shows how many data sets were found on which open data portal. We found ten of 25 data sets on the open data Zurich portal. This is 40% of all found data sets. There are several reasons why the amount of open data Zurich data sets is that high. There are three open data portals which we fully examined, and the one of Zurich is the biggest by far. There is a chance that the open data portals of Switzerland, Germany and Austria provide more useful data sets than listed in the survey table but we did not find them. Another reason why 40% of the found data sets are from open data Zurich is that most of them are available on both portals, the open data Swiss and open data Zurich portal. As mentioned before, if we found a data set on several portals, then the one is annotated which is more specific. If we would not handle it like this, the number of open data Swiss data sets would be higher.

[23] https://developers.google.com/transit/gtfs-realtime/ (accessed 22.5.2017)

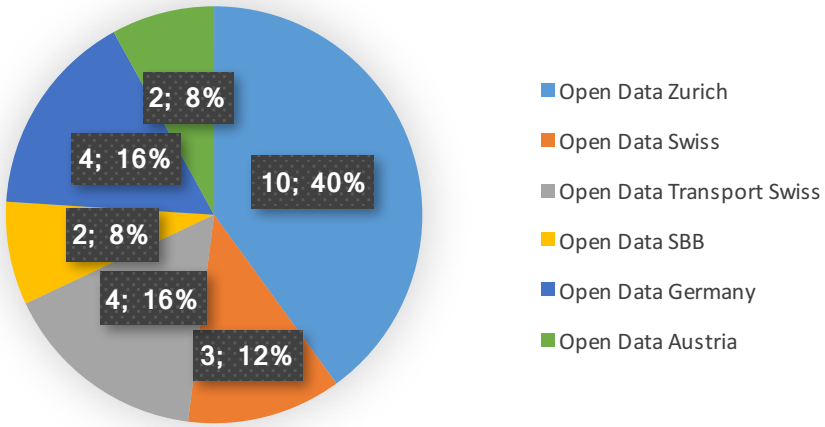

Figure 2: Origin of the data sets

As shown in Figure 3, twelve of 25 data sets hold less than 1000 records. This means, that they contain data about specific objects, like parking spots or weather measure stations. Eight of the other data sets contain much more than 1000 records. Their records mostly include information about public transport. Five of 25 data sets have an unknown number of records. This is because they can only be accessed by API calls. These API calls must contain specific parameters, so not the whole data sets can be fetched.

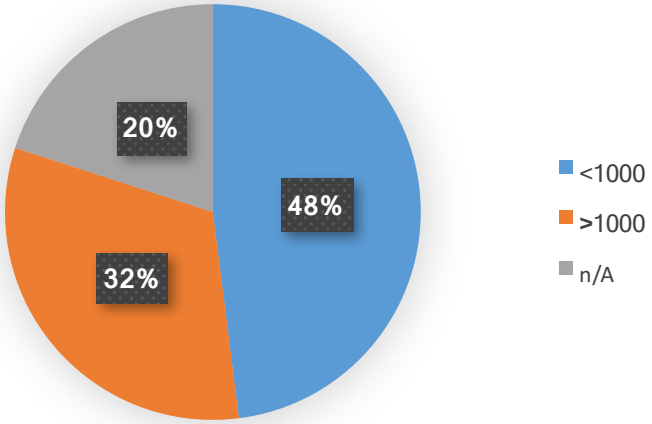

Figure 3: Number of records

Figure 4 shows the publication frequency of the data sets. Eight of 25 (32%) data sets contain continuously updated data. This means that these data are real-time. Five of 25 data sets are published in an interval of ten minutes or less. This means that more than 50% of the found data sets can be fetched every 10 minutes and contain new data which can be streamed out to the Web. Two data sets are listed under the category “others”. These are the data sets *Wassertemperatur Freibäder* and *Echtzeitdaten am Abstimmungstag* because they are update sporadically. Only four of 25 data sets which we considered are updated less frequent than daily. So 19 of 25 data sets (76%) are updated at least once a day.

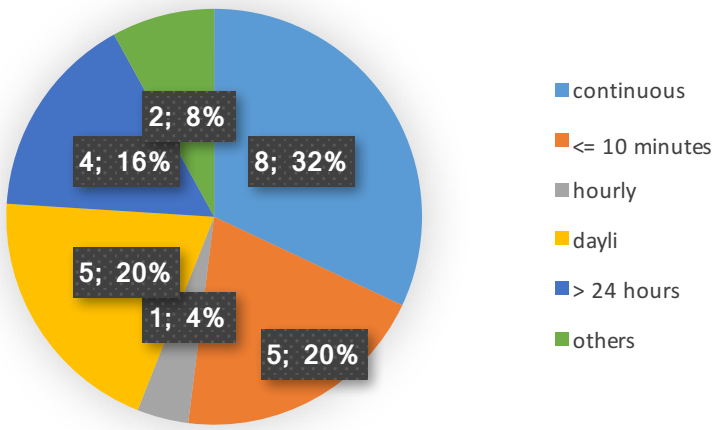

Figure 4: Publication frequency

Figure 5 shows how often the data sets are represented in which format. If a data set can be fetched in different formats, all of them are enumerated in this figure. This means that the number of representations is more than 25, which represents the number of found data sets. The most commonly used data format is CSV. Twelve data sets are available as CSV file. This means that the whole data set can be downloaded. XML is the second most commonly used data format followed by JSON and GeoJSON which is a specific open standard format of JSON. All of the 25 data sets can be fetched in the format of CSV, XML, JSON, GeoJSON or as XML RSS-Feed which are all non-proprietary data formats. In combination with an open license, which all of the found data sets contain, makes them to 3-stars open data, according to the 5-star open data rating system described in Section 3.1. The remaining data formats in Figure 5 occur rarely in the survey table and belong to some particular data sets.

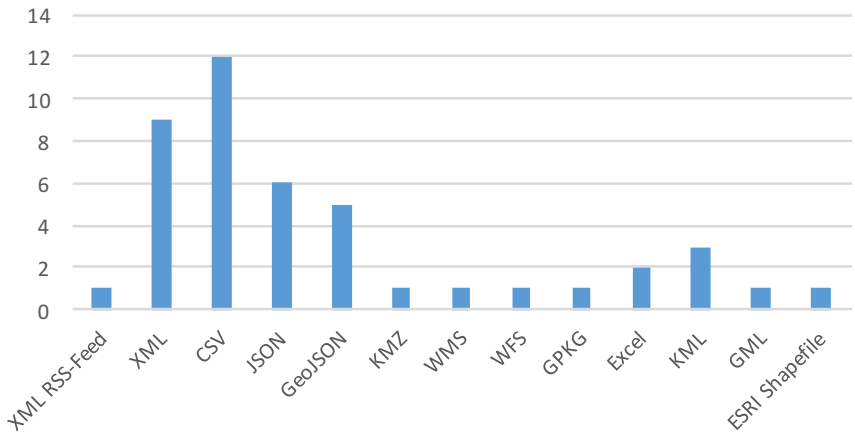

Figure 5: Representation

4 Transforming data into RDF

The goal of this Bachelor Thesis is to increase the number of open data streams on the Web. Additionally, the open data streams should be published according to the Linked Data Principles. This chapter describes the implemented components and their interaction to achieve the goals.

As we described in Chapter 2, TripleWave is a framework which transforms non-RDF streams into RDF streams. This is exactly what we use TripleWave for in this project. We transform and publish a range of OGD sets, among the ones shown in the survey table (see Tables 1-6), according to the Linked Data Principles. TripleWave is able to transform the specific OGD into RDF data using the flexible R2RML mappings. Afterwards TripleWave makes the transformed OGD available through WebSockets.

We do not simply run an individual TripleWave instance for each data set. We have decided to connect the individual TripleWave instances and build a cluster using Apache Kafka. Apache Kafka presents some characteristics which bring advantages for this project.

Vineet & Xia (2017) note that Apache Kafka has high throughput and thus is suitable for processing big data sets. The data sets which are shown in the survey table are either too small or are updated too infrequently to really generate a large data stream. Nevertheless, this project should serve as a guideline for similar projects and therefore we want to cover cases which include large amounts of data. For example, publishing open sensor data generated by large sensor networks can lead to large data streams which Apache Kafka can handle without further problems. As the topic Internet of Things becomes more and more important the number sensor data streams will increase.

As described by Wang et al. (2015), since Apache Kafka can replicate topics among Kafka brokers the reliability of Apache Kafka has increased. The replication greatly reduces the risk of losing data when Kafka brokers crash. Vineet & Xia (2017) conclude that the AMPQ protocol guarantees better reliability than Apache Kafka. But according to Kreps, Narkhede & Rao (2011) Apache Kafka guarantees at-least-once delivery which should be enough for this project. User applications which require exactly-once delivery can achieve that by implementing own data checks. Another advantage of Apache Kafka is the disk storage. Disk storage enables to stream data from the past, if this is required.

The most important characteristics of Apache Kafka are the scalability and modularity. The architecture of Apache Kafka, including Producers, Kafka brokers, topics and Consumers, enables to easily add new elements to the cluster.

We use Kafka Producers to fetch the specific OGD sets, to convert them to JSON, to clean them if required and finally to forward the data to the Apache Kafka brokers. The OGD sets are only available through pull-based mechanisms, either they can be downloaded or they are published through a RESTful API. After Kafka Producers forward the data to the Kafka brokers, each data set is assigned to an own topic. With this approach Apache Kafka offers a simple way of managing data streams. We connect the particular TripleWave instances to the Kafka cluster. This is how the TripleWave instances get the data which should be transformed into RDF. The transformed output data of TripleWave is then again forwarded to the Kafka cluster. Thus the cluster contains both the non-transformed and the transformed data (in different Kafka topics). We implement some push-based mechanisms which stream the data on the Web. This should happen in a way such that it is as easy as possible for clients to access the RDF streams. This approach makes it possible to fetch 3-star level OGD, to transform them and finally publish them as 5-star level OGD.

Figure 6 shows all the implemented and used components. Some of them are shown more than once (for presentation purposes). For example, we illustrate the WebSocketConsumer twice although this component must be run only once.

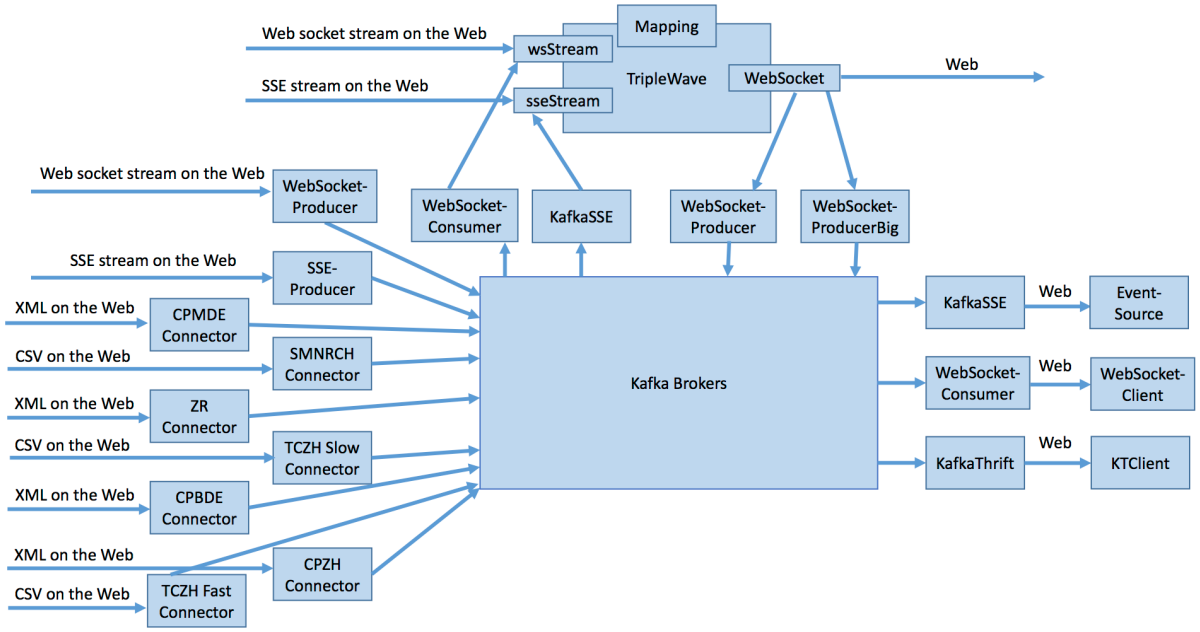

Figure 6: Overview of all implemented components

The two components on the right side called EventSource[24] and WebSocketClient are some example client applications which can be used to connect to the output streams. As the EventSource and WebSocketClient components do nothing else than connecting to a stream and printing it to the console, they are not further explained. They are mentioned in Appendix C.

We did not implement the WebSocket component of TripleWave, since it already existed. But it is included in Figure 6 to show how TripleWave can publish its output stream.

In the Sections 4.1.1 and 4.1.2 we present the components which are responsible for the input of TripleWave and Kafka using WebSockets and SSE/EventSource. Namely these are the SSEProducer, WebSocketProducer, WebSocketProducerBig, sseStream and wsStream components. On the one hand they can be used to fetch existing WebSocket streams and SSE streams and forward them to the cluster. On the other hand, they are used to connect the Kafka brokers and TripleWave.

In Section 4.1.3 we describe the implemented connectors for some specific data sets of the survey table. They all fetch data and forward them to the Kafka brokers using Kafka topics.

In Section 4.2 we present the components which are used to publish the resulting streams. There are three approaches we implement and evaluate. The first component we describe in Section 4.2 is the WebSocketConsumer. This component must be run only once and then handles all incoming requests for connecting to the streams. We use it on the one hand to connect TripleWave and the Kafka cluster and on the other hand to publish the resulting RDF streams. The second component we use is KafkaSSE. This is a library from Wikimedia. We can use it also to connect TripleWave and Kafka and to publish the resulting RDF streams. The third output component is KafkaThrift in combination with KTClient.

In Section 4.3 we introduce the mappings which we created to transform the OGD sets into RDF streams. They are used in the TripleWave.

[24] the code originates from here and we slightly adapted it: https://www.npmjs.com/package/kafka-sse (accessed 23.7.2017)

In Section 4.4 we give an overview of the cluster which we finally run on the University Server. We do not execute all components. Section 4.4 describes which components are excluded and explains the reasons.

4.1 Input Kafka & TripleWave

This chapter describes all the components which feed Apache Kafka and TripleWave with input. These components are on the one hand an important part for connecting the two frameworks Apache Kafka and TripleWave and on the other hand they are used to fetch the open data sets, to clean them where necessary, to convert them to JSON and then to forward the data to the Kafka cluster. This chapter contains three sub sections. Section 4.1.1 describes the components which use WebSockets to connect the Kafka cluster and TripleWave; Section 4.1.2 describes the components which can consume data using Server Sent Events (SSE) and EventSource; and Section 4.1.3 introduces all the components which fetch open data sets using a pull-based approach.

4.1.1 Input using WebSockets

WebSockets are a standard for push-based connections. TripleWave is a WebSocket server where applications can connect to and consume the output of TripleWave. This is how a TripleWave instance is connected to the Kafka cluster. Section 4.1.1.1 describes the component which is responsible for that connection. We must also create a connection in the opposite direction, to feed a TripleWave instance with data from the Kafka cluster. This can also be done using WebSockets and is further described in Section 4.1.1.2.

4.1.1.1 Input Kafka using WebSockets

This section elucidates two components named WebSocketProducer and WebSocketProducerBig. The names of these two components are composed of two parts: WebSocket and Producer. This means, that the components are Apache Kafka Producers which use a WebSocket to get the data. The main purpose of these components is to connect TripleWave to Kafka and thus forwarding the output of TripleWave to a Kafka topic. It is also possible to connect any other data source to Kafka which uses a WebSocket connection. The difference between these two components is that WebSocketProducer works good for data streams with low throughput and WebSocketProducerBig is designed to work for data streams with high throughput. As the most data sets presented in the survey table (Tables 1-6) are small data sets or their update frequency is low, the WebSocketProducer normally suffices. But in Appendix B we show how to run the cluster for a high throughput data stream. There we need the WebSocketProducerBig component. First we describe the WebSocketProducer component and after that we introduce the WebSocketProducerBig component.

This pseudo-code illustrates how the WebSocketProducer works:

```
Require: Address of TripleWave and a Kafka topic T
1:      Initialize new Kafka Producer P;
2:      Wait until P is ready then:
3:              Create new WebSocket client C;
4:              Connect C to TripleWave;
5:              On new incoming data from TripleWave:
6:                      Stringify incoming data;
7:                      Forward incoming data to the Kafka topic T;
```

The WebSocketProducer uses the kafka-node library[25] to create a Kafka Producer and connect it to the Kafka cluster. For the WebSocket connection, the Primus library[26] is used. Primus can use several real-time frameworks for establishing a connection. As TripleWave uses Primus in combination with the ws library[27] the same is done in the WebSocketProducer.

The output of TripleWave is in the JSON-LD format. Thus the input format of the WebSocketProducer is JSON. As the Kafka cluster takes as input only plain text, the JSON input must first be stringified before sending it to Kafka. If the WebSocketProducer is used for connections to other sources than TripleWave, the format of the data should be JSON, otherwise the code must be adapted.

WebSocketProducer takes four command line parameters to ease the handling of the configuration:

- –t=topic for determining the topic name to which the producer should send the data, e.g.: –t=test2
- –u=URL which consists of the base URL of TripleWave, e.g.: –u=ws://localhost:4040
- –p=pathname of TripleWave, e.g.: –p=/triplewave/primus
- –q=query which contains a URL query e.g.: –q=/?variable=value

If they are omitted, the producer takes the given default parameters in the code. Except for parameter q, the default parameters in the code match the above presented example parameters. The default value of the parameter q is an empty string, as no query element is needed to connect TripleWave.

This pseudo-code illustrates how the WebSocketProducerBig works:

```
Require: Address of TripleWave and a Kafka topic T
1:      Initialize new Kafka Producer P;
2:      Wait until P is ready then:
3:              Create new WebSocket client C;
4:              Connect C to TripleWave;
5:              On new incoming data from TripleWave:
6:                      Stringify incoming data;
7:                      Push the data on the payloads;
8:              After one second, send payloads to Kafka topic T and reinitialize them
```

The WebSocketProducerBig works similar to the WebSocketProducer. Instead of forwarding each message individually to Kafka, it collects the incoming messages and once a second sends them to Kafka in one batch. Doing so reduces the traffic between Kafka Producer and Kafka brokers drastically. The WebSocketProducer and the WebSocketProducerBig components accept the same parameters.

4.1.1.2 Input TripleWave using WebSockets

wsStream is a WebSocket stream connector necessary for TripleWave[28]. TripleWave already has some example Web stream connectors implemented. One of them, named wikiStream2, uses an old version of a WebSocket (it uses the *socket.io*[29] 0.9.1). So we decided to implement a connector, named wsStream, which uses an updated version of WebSockets (*Primus* 7.0.2 in combination with *ws* 1.1.0). This connector establishes a connection to a WebSocketConsumer (described in

[25] https://www.npmjs.com/package/kafka-node (accessed 17.7.2017)

[26] https://github.com/primus/primus (accessed 17.7.2017)

[27] https://www.npmjs.com/package/ws (accessed 17.7.2017)

[28] http://streamreasoning.github.io/TripleWave/docs.html#webstream (accessed 17.7.2017)

[29] https://www.npmjs.com/package/socket.io (accessed 17.7.2017)

Section 4.2.1). With the approach, we can establish a connection from Kafka's output to TripleWave's input.

We started from wikiStream2. We changed only the WebSocket client and the information needed (request URL) to connect to the Kafka cluster. TripleWave already uses the Primus library, so we decided that the connector wsStream implements a WebSocket client using Primus in combination with the ws library.

To connect a WebSocket client to the component called WebSocketConsumer (described in Section 4.2.1), the base URL, the pathname and a URL query are needed. For further details of how to use wsStream, see Appendix B. The required information is defined in the first lines of the code of wsStream. There is no functionality implemented to define them using command line parameters.

When a connection is established, wsStream gets data from the Kafka cluster and simply forwards it to TripleWave. As the output of the Kafka cluster is parsed to JSON and as TripleWave can handle JSON input no further parsing is required.

We have to note one more important fact: the WebSocketConsumer uses Primus 7.0.2. TripleWave 2.1.1 however uses Primus 6.0.1. There have been made some breaking changes from Primus version 6 to Primus version 7 including a new implementation of the heartbeat mechanism[30]. Thus these two versions do not work together if the heartbeat mechanism is not disabled. Because a heartbeat mechanism is an important part of a connection, we do not disable it. So it is necessary to update TripleWave, so that it uses at least Primus 7.0.0. Otherwise the connection to the WebSocketConsumer will not work correctly.

4.1.2 Input using SSE/EventSource

Wikimedia maintains a real-time stream which contains all changes in a machine-readable format made to MediaWiki[31]. This stream recently adapted the EventSource interface[32]. Motivated by this stream, we implement an input component for Kafka (described in Section 4.1.2.1) and a Web stream connector for TripleWave (described in Section 4.1.2.2), which use the EventSource interface to consume Server Sent Events.

4.1.2.1 Input Kafka using SSE/EventSource

This section introduces the SSEProducer component. It consists mainly of a Kafka Producer and the eventsource library[33]. The SSEProducer enables to connect any SSE Web stream to the Kafka cluster. As mentioned above, it is motived by the Wikimedia stream, which is available as SSE Web stream.

The following pseudo-code illustrates the mechanism of this component:

```
Require: Address of a SSE Web stream and a Kafka topic T
1:      Initialize new Kafka Producer P;
2:      Wait until P is ready then:
3:              Connect to SSE Web stream;
4:              On new incoming events:
5:                      Forward the message part of the event to the Kafka topic T;
```

[30] https://github.com/primus/primus/releases/tag/7.0.0 (accessed 17.7.2017)

[31] https://www.mediawiki.org/wiki/Manual:RCFeed (accessed 17.7.2017)

[32] https://wikitech.wikimedia.org/wiki/EventStreams (accessed 17.7.2017)

[33] https://www.npmjs.com/package/eventsource (accessed 17.7.2017)

It is quite simple to connect a JavaScript application to a SSE Web stream, which is one advantage of this protocol. An incoming Server Sent Event consists of three objects: event, id and data[34]. The data object contains the actual message we are interested in. Thus the SSEProducer only forwards the data object to the Kafka cluster. This can easily be changed if needed. In contrast to the WebSocketProducer (see Section 4.1.1.1), this component does not parse the incoming stream because the SSE Web stream delivers plain text. The Wikimedia real-time Web stream publishes its data as stringified JSON, which is exactly the required format for our Kafka cluster. If a SSE Web stream should be connected to the Kafka cluster, which contains a different format, a parsing algorithm must be implemented.

The SSEProducer can take two command line parameters:

- –t=topic, for determining the topic name to which the producer should send the data, e.g. –t=test
- –u=URL, which contains the URL of the SSE Web stream, e.g. –u=https://stream.wikimedia.org/v2/stream/recentchange

If they are omitted, the producer takes the given default parameters in the code which match the above presented example parameters.

4.1.2.2 Input TripleWave using SSE/EventSource

In the preceding section we introduced a component which can connect a SSE Web stream to the Kafka cluster. For the same reasons, we implement a similar component for TripleWave, named sseStream. With this extension it is possible to connect TripleWave to a SSE Web stream, for transforming this stream into an RDF stream. The sseStream component is a Web stream connector for TripleWave.

sseStream works similar to SSEProducer (introduced in Section 4.1.2.1). sseStream also uses the eventsource library. Only a URL is required to determine the source of the SSE Web stream. When the connection is established, the incoming event data objects are converted to JSON and then forwarded to TripleWave for the transformation. In contrast to the SSEProducer, the sseStream converts the incoming stringified JSON to JSON, as TripleWave requires the input stream in JSON. Again, if the SSE Web stream contains a different format as stringified JSON, a parser mechanism must be implemented before forwarding the data to TripleWave.

The sseStream component can also be used to connect TripleWave to the Kafka cluster. This is enabled by the kafka-sse library[35], which is published by Wikimedia. For a short introduction into this library see Section 4.2.2.

4.1.3 Input through fetching data sets

This section introduces the implemented connectors which fetch, clean, parse and forward the OGD sets to the Kafka cluster. As shown in Figure 5 (Section 3.2.3), CSV and XML are the two most common data representations in the survey table. Thus we focused on writing connectors for these data formats. All the data sets mentioned in the survey table have their own characteristics and differ in various aspects. Therefore, it is not possible to write one connector which fits on all the data sets in the survey table. Specific, ad-hoc connectors are therefore written for fetching specific data set. We chose the data sets, such that we can write connectors which cover as many characteristics as possible and, therefore, can be seen as reference point for further implementations. Table 7 gives an overview over the implemented connectors and their characteristics.

The first important characteristic of the data sets in the survey table is the distinction between data sets that replace all their content at each update and data sets that append new content at each update. As example for the former case, we can mention SMNRCH (see Tables 1-6), which contains the measured

[34] https://wikitech.wikimedia.org/wiki/EventStreams (accessed 17.7.2017)

[35] https://www.npmjs.com/package/kafka-sse (accessed 17.7.2017)

rainfall for about 100 weather stations in Switzerland. This data set is updated every ten minutes and at each update all the old data are replaced by the current one. We call this type of data sets *non-appending*. As example for the latter case, let's consider the data set which counts the traffic in Zurich (TCZH, see Tables 1-6), and is published weekly. The new content is appended to the existing one, so the data set grows weekly. We call this type of data sets *appending*. This is important because it is not desired to stream all the old data at every time the data set is changed. This characteristic is reported in column 3 in Table 7.

Appending data sets require filters to distinguish old and new data. Also, *non-appending* data sets may require filters, e.g. data sets where not all entities are updated at the same time or where not all entities own new values between two updates. With filter functions we ensure that the stream only contains values which have changed. Column 4 in Table 7 represents this characteristic.

Another aspect which we must consider is the size of the data set. There are data sets which contain around 100 records and others with several hundred thousands or even millions of records. The TCZH Connector (see Section 4.1.3.2) fetches a data set which contains now (June 2017) more than 500'000 records and at the end of the year will contain more than one million records. The ZR Connector (see Section 4.3.4) fetches a data set with eight records. So the amount of data which is forwarded to Kafka can vary. The Producer from the kafka-node library does not automatically handle the batch size of its sent messages. This is important because for every sent message, there is an acknowledge from the Kafka broker. When there is only a low number of entities in the data set or when the data are forwarded to Kafka in a manageable speed, there is no problem to send every element separately to the Kafka cluster. But fetching half a million records and sending them separately as fast as possible to the Kafka cluster leads to an overload because of all these acknowledgments. This characteristic is reported in column 5 in Table 7. The corresponding sections about the connectors report how the connectors deal with this aspect.

Another difference is if and how the content must be cleaned and enriched to be correctly converted to RDF. This is also managed in the connectors before the data are forwarded to the Kafka cluster.

The column *Other* in Table 7 reports if there is anything special implemented in the corresponding connector.

Section	Format	Append	Filter	Size	Other
4.1.3.1 SMNRCH Connector	CSV	No	No	Small	-
4.1.3.2 TCZH Slow Connector	CSV	Yes	Yes	Big	Slow processing
4.1.3.3 TCZH Fast Connector	CSV	Yes	Yes	Big	Fast processing
4.1.3.4 CPZH Connector	XML	No	Yes	Small	RSS Feed
4.1.3.5 CPBDE Connector	XML	No	Yes	Small	Conditional Get
4.1.3.6 ZR Connector	XML	No	Yes	Small	Decoding corrected
4.1.3.7 CPMDE Connector	XML	No	Yes	Small	-

Table 7: Overview Connectors

4.1.3.1 SMNRCH Connector

We explain in this section the connector written for the data set SMNRCH (see Tables 1-6). This data set contains the measured rainfall for the last ten minutes for about 100 weather stations in Switzerland. We decided to write a connector for this data set to give an example for a CSV & non-appending data set. The data set contains always exactly one measurement value for each weather station, namely the most current value. No additional examination should be done to distinguish between new and old data, when fetching the data set. (However, this may be required, when the goal is to stream only changed values, e.g. the rainfall value has changed for a certain station in the last ten minutes.)

The SMNRCH Connector makes use of mainly three packages. First a Kafka Producer is needed to be able to forward the fetched data to the Kafka cluster. We use for this the kafka-node library. Second, the request library[36] is used to make an HTTP request for fetching the data set. And the third package is the CSV to JSON parser[37], which we need to convert the data to JSON for TripleWave.

The following pseudo-code illustrates how the connector works:

```
Require: a Kafka topic T
1:      Initialize new Kafka Producer P;
2:      Wait until P is ready then:
3:              Make HTTP request to fetch the data set;
4:              Remove the first two lines;
5:              Parse CSV to JSON;
6:              On every parsed line:
7:                      Stringify JSON record R;
8:                      Send R to the Kafka topic T;
9:      Repeat every ten minutes;
```

There are some notes:

- Line 4: the first two lines must be removed, as they do not contain required data.
- Line 8: each line thus each record is sent separately to the Kafka cluster. As the data set contains only about 100 records, this is not a problem.
- Line 9: as the data set is updated every ten minutes, we set the interval to ten minutes.

It is possible to define the Kafka topic to which the data should be sent over a command line tool, through parameter –t=topic.

It is not possible to write one single connector for all CSV data sets also because the data sets are not consistently formatted. There is sometimes information in the data file, which should not be streamed or even disturb correct parsing (in this example the first two lines).

4.1.3.2 TCZH Slow Connector

The TCZH Slow Connector fetches the TCZH data set (see Tables 1-6), which contains the number of pedestrians and cyclists passing certain spots in Zurich. The numbers are update every 15 minutes but the data set is published only once a week. The new data records are appended to the existing ones, thus the data set continually grows until the end of the year, then a new data set is started. Under the corresponding link to this data set, there are several data sets listed, one for every year since 2009. Because the data sets contain measurements for several stations every 15 minutes, the number of records at the end of the year exceeds one million. Around 25'000 records are added every week. Because of the size of the data, we must ensure that data set is transformed and streamed out slowly enough to not overload the WebSocket (or SSE/EventSource) output stream. This can happen as Kafka can handle more data than a simple WebSocket or SSE/EventSource connection.

The connector uses the same three main components as the SMNRCH Connector, described in Section 4.1.3.1, namely the kafka-node library, the request library and the csvtojson library. The work-flow is different, as illustrated in the following peuso-code:

[36] https://www.npmjs.com/package/request (accessed 17.7.2017)

[37] https://www.npmjs.com/package/csvtojson (accessed 17.7.2017)

```
Require: Address of a TCZH data set and a Kafka topic T
1:    Initialize new Kafka Producer P;
2:    Wait until P is ready then:
3:        Make HTTP request to fetch the data set;
4:        Remove the first line;
5:        Seperate new data from old data;
6:        If there is new data:
7:            Convert one line to JSON;
8:            Stringify JSON record;
9:            Send record to the Kafka topic T;
10:       Repeat after 20 milliseconds;
11:   Repeat after 3.5 days;
```

As this data set also has its individual characteristics, the connector was adapted so that it fits the requirements:

- Line 4: after fetching the data set, the first line, which includes the headers, is removed, as the headers are added manually using the options of the csvtojson library.
- Line 5: the old data are separated from the new ones. As mentioned above, this must be done because the data set contains the measurements of the current year and it is not useful to stream all of them every week.
- Lines 6-10: if there is new data available, then the component processes line by line with a pause of twenty milliseconds. We take a relatively slow data process rate so that as many WebSockets connections as possible can handle the stream. With this process rate, there are no problems according to the traffic between Kafka Producers and Kafka brokers although we use in this approach a batch size of one.
- Line 11: the data set is published once a week. Because we do not know if it is always exactly one week, we decided to fetch it twice a week to ensure we do not miss new updates.

The connector takes two command line parameters: one for the topic (-t=topic) and one for the URL (-u=URL). There is a command line parameter available for the URL, because there are individual data sets available for every year (since 2009). So using the command line parameter, it is easier to define which URL the connector should fetch. The default data set is the one of the year 2017.

4.1.3.3 TCZH Fast Connector

This component fetches the TCZH data set, like the TCZH Slow Connector does (see Section 4.1.3.2). The TCZH Fast Connector can scale exploiting the cluster. Thus, this component should fetch, transform and forward the data as fast as possible. In this section, we describe how the TCZH Fast Connector works; in Appendix B we give an example of how to use Kafka and TripleWave in combination with this connector.

This component uses the same libraries as the TCZH Slow Connector. The following pseudo-code shows how the TCZH Fast Connector works:

```
Require: Address of a TCZH data set and three Kafka topics T0, T1, T2
1:    Initialize new Kafka Producer P;
2:    Wait until P is ready then:
3:        Make HTTP request to fetch the data set;
4:        Remove the first line;
5:        Separate new data from old data;
```

```
6:          If there is new data:
8                   Rotate between three topics;
9:                  Convert 5000 lines to JSON;
10:                 Stringify JSON records;
11:                 Send records to one of the three the Kafka topics T0, T1 and T2;
12:         Repeat as long as not all data are processed;
13:   Repeat after 3.5 days;
```

Lines 6-12 describe the differences compared to the TCZH Slow Connector. Instead of processing one record every 20 milliseconds, this connector processes the data as fast as possible. If new data are available, the TCZH Fast Connector parses 5000 records to JSON at the same time and then forwards them to a Kafka topic. The important part is that this connector rotates between three Kafka topics after each batch with the size of 5000 records. This means that the data set are finally split into three subsets. Doing so, we can connect three TripleWave instances to the Kafka brokers and thus transform data into RDF in parallel.

The TCZH Fast Connector takes four parameters. Three for the three topics (-t0=topic0; -t1=topic1; -t2=topic2) and one parameter for the URL, similar to the TCZH Slow Connector.

4.1.3.4 CPZH Connector

The CPZH Connector fetches the CPZH data set (see Tables 1-6) which contains the number of free slots of Zurich's parking facilities. This data set is available as XML RSS-Feed and contains real-time data. As this is the only one data set in the survey table which contains an XML RSS-Feed we have written a connector for it, to give a guideline for this case.

The CPZH Connector uses mainly four packages to fetch, filter, convert and forward the data set to a Kafka topic. The kafka-node library and the request library are used for initializing a Kafka Producer and fetching the data set. For XML RSS-Feeds, there is available a library called feedparser[38] which can parse the XML data to JSON. The fourth package is the stream library[39], which is available from Node.js itself. It is needed to forward the fetched data set to feedparser.

The work-flow of the connector is described in the following pseudo-code:

```
Require: a Kafka topic T
1:      Initialize new Kafka Producer P;
2:      Wait until P is ready then:
3:          Make HTTP request to fetch the data set;
4:          If data set contains new data:
5:              Separate old data from new data;
6:              Stream new data into converter;
7:              Convert new data to JSON;
8:              Enrich data;
9:              Clean data;
10:             Stringify data;
11:             Send new data to the Kafka topic T;
12:     Repeat after 10 seconds;
```

[38] https://github.com/danmactough/node-feedparser (accessed 17.7.2017)

[39] https://nodejs.org/api/stream.html (accessed 17.7.2017)

Similar to the other connectors, first a Kafka Producer is initialized and when it is ready, the connector starts to fetch the data set. Following steps are explained in detail:

- Line 4: the fetched data set is compared to the one which was fetched ten seconds ago. If the two data sets are identical, the rest of the code is not executed.
- Line 5: otherwise, new data items are separated from the old one. Thus, only the changed data items are forward to the Kafka cluster. This separation function is adjusted to the structure of the CPZH data set. It will not work properly if the connector is used to fetch other XML RSS-Feeds, as the structures of the feeds may differ.
- Line 6: for forwarding the new data items to the parser, a stream object is needed.
- Lines 8-9: these two steps are important for transforming the OGD set into RDF, which is done in TripleWave. TripleWave needs some additional attributes to correctly apply the corresponding R2RML mapping (see Section 4.3.2). We add the required attributes in these two steps. Especially we add links to entities from LinkedGeoData[40] to link the data set to other data.
- Line 12: the last step which we explain is that every ten seconds the data set is fetched and the whole process restarts from Line 3. The OGD portal of Zurich mentions that the CPZH data set is real-time respectively continuously updated. After implementing and running the connector, it came out that the data set is only updated roughly once a minute. We therefore set the repeating interval to 10 seconds, to limit the number of calls.

The CPZH Connector can take one parameter to define the topic to which the data should be forwarded. This is done by using the –t=topic parameter in the command line.

4.1.3.5 CPBDE Connector

This connector fetches the CPBDE data set (see Tables 1-6) which contains the number of free parking spaces of parking facilities in the city of Bonn. The special characteristic of this data set is that it is published applying the conditional GET. This is a mechanism where the data set is sent from the server to the client only when it has changed since the last request therefore reducing the traffic in the network. This is done by adding some specific headers to the request[41]. The CPBDE Connector implements the requirements for the conditional GET and thus can be seen as a guideline for fetching similar data sets.

The CPBDE Connector requires three packages: the request library for fetching the data set, the kafka-node library for instantiating a Kafka Producer and the xml2js library[42] to convert XML data to JSON.

The following pseudo-code illustrates the mechanism of the CPBDE Connector:

```
Require: a Kafka topic T
1:      Initialize new Kafka Producer P;
2:      Wait until P is ready then:
3:          Make HTTP request to fetch the data set;
4:          If response status == 200:
5:              Parse data from XML to JSON;
6:              Separate new data items from old ones;
7:              Enrich the data;
8:              Send stringified, new data to the Kafka topic T;
9:          Else if response status == 304:
```

[40] http://linkedgeodata.org/About (accessed 28.6.2017)

[41] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html (accessed 17.7.2017)

[42] https://github.com/Leonidas-from-XIV/node-xml2js (accessed 17.7.2017)

```
10:                 Prepare variables for next call;
11:     Repeat after 10 seconds;
```

The following steps are explained in more details:

- Line 3: an HTTP conditional GET request is done. Such a request contains additional headers called If-Modified-Since and If-None-Match. These two headers are sent with the request to the server. The server uses them to find out if the data on the server have changed since the last request. This means that these two headers must be updated for every call.
- Lines 4-6: if the server detects that the data have changed since the last request, the response contains the status 200 and the data set. This data set is in XML, so it must be converted to JSON. As only the changed data items should be forwarded to Kafka, they must first be separated.
- Line 7: for the transformation into RDF, some additional attributes are required, namely links to entities on LinkedGeoData and URLs to the parking Web site. They are added in this step.
- Line 9: if the server detects that the data set has not changed since the last call, the response contains the status 304. In this case, the response does not contain the data set.
- Line 11: the fetching loop is repeated every ten seconds. Although the CPBDE data set is labelled as real-time, it seems that it is not updated every second but rather every one to two minutes. Therefore, the fetching loop is set to ten seconds to limit the number of calls.

The CPBDE Connector can take one parameter to define the topic to which the data should be forwarded. This is done by using the –t=topic parameter in the command line.

4.1.3.6 ZR Connector

This connector fetches the ZR data set (see Tables 1-6) which contains information about the bicycle rental in Zurich. This data set also belongs to the real-time data sets in the survey table and therefore is interesting for being published on the Web as RDF stream.

As the data set must be fetched and forwarded to the Kafka cluster, the request library and the kafka-node library are used. The ZR data set is available as XML and thus the xml2js library is required to convert the XML data to JSON.

This pseudo-code mirrors how the ZR Connector works:

```
Require: a Kafka topic T
1:      Initialize new Kafka Producer P;
2:      Wait until P is ready then:
3:              Make HTTP request to fetch the data set;
4:              Parse the XML data to JSON;
5:              Clean data;
6:              Separate new data items from old ones;
7:              Enrich the remaining data;
8:              Forward stringified data to the Kafka topic T;
9:      Repeat after 3 seconds;
```

This connector is quite simple compared to the previous one:

- Lines 3-4: when the Kafka Producer is ready, the HTTP request is sent to the server. The data set contained in the response is converted to JSON.
- Line 5: one special thing about this data set is that there are decoding errors, which did not happen in the other data sets. These are corrected automatically by the connector.

- Line 7: the records must be enriched so that they contain the necessary attributes for the R2RML transformation done in TripleWave. Time stamps and links to entities from LinkedGeoData are added in this step. Additionally, the X and Y coordinates are added.
- Line 9: the ZR data set contains real-time data according to the OGD portal of Zurich. In contrast to the CPBDE data set (see Section 4.1.3.5) and to the CPZH data set (see Section 4.1.3.4) no fixed update rate was identified. So the loop interval is set to three seconds, to minimize the risk to miss some updates.

The ZR Connector can take one parameter to define the topic to which the data should be forwarded. This is done by using the –t=topic parameter in the command line.

4.1.3.7 CPMDE Connector

This connector fetches the CPMDE data set (see Tables 1-6) from the survey table. It contains data about parking facilities of the city of Moers. As this data set belongs to the real-time data sets in the survey table we decided to transform it into an RDF streamed and publish it on the Web. As every data set has its own characteristics we had to write a separate connector for fetching, editing and converting the CPMDE data set.

We do not explain the whole workflow of this connector, because the CPMDE and the ZR data sets can be fetched similarly. Section 4.1.3.6 contains more information about the workflow. We slightly adapted the ZR Connector. We removed the decoding error correction, as it is not required. Additionally, the CPMDE Connector deletes an attribute, because we are not interested in it and it would influence the separation of new and old data. The third thing we changed is the content of the enrichment method, because two different data sets require two different enrichment methods. Links to entities on LinkedGeoData and time stamps are added to the data records.

4.2 Output Kafka

In this section, we describe the implemented components which make it possible for clients to connect to the transformed output Web streams. We implement push-based mechanisms, so that the clients automatically get the new data when the OGD sets are updated. There are three mechanisms which we implemented. In Section 4.2.1, we describe how we enable to connect to these data streams using WebSockets. Section 4.2.2 gives a short introduction about how we publish the data streams using SSE/EventSource. We explain in Section 4.2.3 how we use Apache Thrift[43] as alternative to JSON-LD to serialize the streams.

4.2.1 Output Kafka using WebSockets

WebSockets are a standard for push-based connections and thus we implement a component named WebSocketConsumer which makes it possible for clients to connect to the RDF output streams using this protocol. This component supports a simple way to connect to the streams, thus supporting usability which is important to increase the benefit of the OGD RDF streams.

As the name suggests, the WebSocketConsumer consists of two entity types: A WebSocket server and a Kafka Consumer Group. We use the Primus library to maintain the WebSocket server. By using the same WebSocket library in every component where one is needed, we ensure the compatibility. To set up Kafka Consumer Groups we use the kafka-node library.

The following pseudo-code illustrates how the WebSocketConsumer works:

```
1:    Initialize new WebSocket server;
2:    Get available topics;
```

[43] https://thrift.apache.org/ (accessed 26.6.2017)

```
3:      On new request:
4:          Read the desired topics out of the request (if available);
5:          If the desired topics are not valid:
6:              Reject the connection;
7:          Else:
8:              Initialize new Consumer Group with desired topics or default topic;
9:              On new message from Kafka topics:
10:                 Forward it to the WebSocket client as JSON;
11:     On WebSocket client disconnects:
12:         Close the Consumer Group;
```

We explain the following steps in more details:

- Line 2: the available topics are all topics which are already created in the Kafka cluster. The available topics are refreshed once an hour. So if the Kafka cluster gets new topics including new streams, they are at the latest available for clients after one hour. With this approach, it is not necessary to restart the WebSocketConsumer after adding new topics.
- Line 4: we implemented the functionality that a WebSocket client can connect to more than one topic at the same time, if that is desired. The topics can be given in the query of the request. For further details of how a user can connect to the Kafka cluster using WebSockets, see Appendix C.
- Line 6: we noticed that if the topics in the query do not already exist, then they are created when initializing a new Consumer Group. To prevent that wrong requests can create topics, the request is first checked; if the topics are not available, the request is rejected.
- Line 10: the WebSocketConsumer parses every message from the Kafka cluster to JSON before streaming it to the client. This is necessary because the Kafka cluster processes plain text.
- Line 12: if a client disconnects from the WebSocketConsumer, the corresponding Kafka Consumer Group will be closed. Otherwise the Consumer Group would continue running although the WebSocket connection does not exist anymore.

4.2.2 Output Kafka using SSE/EventSource

We have decided to give clients the possibility to consume the output Web streams using the EventSource interface. There is an available package, implemented by Wikimedia, called kafka-sse[44]. It enables clients to easily connect to Kafka topics using SSE/EventSource. An example of how to connect is available in Appendix C. We decided to support this format because it works on the top of HTTP, and it is a recent effort for creating Web streams. Also the most browsers support SSE/EventSource.

4.2.3 Output Kafka using Apache Thrift

Slee, Agarwal & Kwiatkowski (2007) describe several advantages of Apache Thrift and why it was built. There is one advantage we wanted to look closely: The Serialization & Deserialization functionality and thus its ability to efficiently transmit messages. RDF Binary[45] is an approach which describes a format to encode RDF using Apache Thrift. We examined if we can reuse RDF Binary for RDF streams.

[44] https://www.npmjs.com/package/kafka-sse (accessed 26.6.2017)

[45] https://afs.github.io/rdf-thrift/ (accessed 26.6.2017)

Apache Thrift implements a client-server model. This approach is not designed to send data from the server to the client in a push-based way. The client part sends data to the server and the server responds to that request. Thus a more complicated approach is necessary, because the Apache Thrift server must be run on the client side (i.e. where the data should be streamed to) and the Apache Thrift client must be implemented in combination with a Kafka Consumer to forward the RDF stream. This stands in contrast to the structure of push-based mechanisms, where the server sends the data to the client. The following lines illustrate how the both parts interact to establish a connection. (C) stands for what the client side makes (not the Apache Thrift client but the client who wants to consume the RDF stream, called KTClient in Figure 6) and (KT) stands for what the KafkaThrift component makes:

1: (KT) Create an HTTP server and listen for incoming request;
2: (C) Create an Apache Thrift server and listen for incoming streaming data;
3: (C) Make an HTTP request which includes its own IP address and port;
4: (KT) On request:
5: (KT) Create an Apache Thrift client which connects to the Apache Thrift server corresponding to the IP address and port in the request;
6: (KT) As the connection is established, create an Apache Kafka Consumer Group;
7: (KT) On new message in the Consumer Group:
8: (KT) Parse the message to JSON;
9: (KT) Parse the JSON message to NQuads;
10: (KT) Create the defined Apache Thrift object using the NQuads;
11: (KT) Transmit the NQuads to the Apache Thrift Server;
12: (C) On incoming message:
13: (C) Parse the Apache Thrift object back to NQuads;
14: (C) Parse the NQuads to JSON

The following steps are explained in more details:

- Line 1: must be done only once, because the created HTTP server will then handle all incoming requests.
- Line 2: the Apache Thrift client sends messages to an Apache Thrift server and thus it must be run where the stream is consumed. The client application must first start the Apache Thrift server before an Apache Thrift client can be started and connected to transfer the streaming data.
- Line 3: IP address and port are required by the Apache Thrift client as it must know where to connect to and send the streaming data.
- Lines 8-11: the RDF Binary format describes RDF streams row by row, which means that the Apache Thrift client must send each RDF Quad individually. As TripleWave outputs the data as JSON-LD the Apache Thrift client must do several parsing steps before it is able to send the data to the Apache Thrift Server.
- Lines 13-14: the Apache Thrift server then must parse the incoming data back to JSON-LD which requires again several steps.

There come several disadvantages with this approach. We noticed that multiple parsing steps must be done before the Apache Thrift client is able to forward the streaming data to the Apache Thrift server. These multiple parsing steps require an individual instance for each OGD set because the data objects are different. Another disadvantage of this approach is that the client application must run an Apache Thrift server and thus must use a free port for the incoming message. This could lead to Firewall issues.

We have implemented such an approach for one specific data stream to be able to evaluate the transmission. We wondered if the RDF Binary scheme is more efficient concerning the package size. So we

evaluated the package size and compared it to the package size when using WebSocket or SSE/EventSource. We present the results of this evaluation in details in Section 5.4.

4.3 Increasing the number of stars

All the connectors we mentioned in Section 4.1.3 fetch a specific data set. Then each data set is forwarded to the Apache Kafka cluster and to TripleWave, where it is transformed into RDF, and then streamed back to the cluster. Through the transformation into RDF we increase the number of stars for the OGD sets according to Tim Berners-Lee 5-star open data model. We keep as much information as possible from the original data sets, so that the resulting RDF streams contain as much data as possible. We describe in this chapter the mappings we use to transform the OGD data into RDF. In the following sections we use prefixes to describe the mappings:

- dex: http://data.example.com/TripleWave-transform/
- ex: http://example.com/terms#
- dtx: http://vocab.datex.org/terms#
- rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
- owl: https://www.w3.org/2002/07/owl#
- ldg: http://linkedgeodata.org/
- purl: http://purl.org/dc/terms/
- geo: http://www.w3.org/2003/01/geo/wgs84_pos#
- rdfs: http://www.w3.org/2000/01/rdf-schema#

4.3.1 TCZH Mapping

We select this data set to fetch, transform and stream it out because of its size. So we want to show how this can be done with large data sets and what the challenges are. Because we have not found a suitable vocabulary for the TCZH data set in appropriate time, we use example URIs for transforming it into RDF. The main challenge in managing this data set is its high throughput. Table 8 shows which triples TripleWave creates using the mapping, illustrated for one record of the data set.

Subject	**Predicate**	**Object**
dex:4688617	ex:position	U15G3063866
dex:4688617	purl:date	2017-01-22T14:30:00
dex:4688617	ex:bicyclesIn	""
dex:4688617	ex:bicyclesOut	""
dex:4688617	ex:pedestriansIn	16
dex:4688617	ex:pedestriansOut	11

Table 8: TCZH Mapping

We create a triple for each column of the record. It is normal that not all columns have values, because not each station counts the number of bicycles and the number of pedestrians passing the location. So not every object of a triple has a value.

4.3.2 CPZH Mapping

We select the CPZH data set because it is updated frequently and contains information about 37 parking facilities in Zurich, which is a big number. There is a vocabulary named datex[46] which has been created for describing parking facilities. So we reuse it as much as possible. We also found an ontology defining a parking facility from LinkedGeoData[47]. For every parking facility in the data set, we searched the corresponding entity on OpenStreetMap[48] to link the data set to LinkedGeoData entities. Doing so we are able to raise the CPZH data stream to the 5-star level of open data. Table 9 shows the triples which TripleWave creates with our mapping.

-

Subject	Predicate	Object
dex:Parkgarage am Central	dtx:parkingName	Parkgarage am Central
dex:Parkgarage am Central	dtx:address	Seilergraben
dex:Parkgarage am Central	dtx:urlLinkAddress	http://www.plszh.ch/park-haus/central.jsp?pid=central
dex:Parkgarage am Central	dtx:parkingMode	open
dex:Parkgarage am Central	dtx:numberOfVacantParking-Spaces	0
dex:Parkgarage am Central	dtx:dateTime	2017-06-28T13:39:57.000Z
dex:Parkgarage am Central	rdf:type	ldg:ontology/Parking
dex:Parkgarage am Central	owl:sameAs	ldg:geometry/node460419758

Table 9: CPZH Mapping

4.3.3 CPBDE Mapping

We select the CPBDE data set for its update frequency similarly to the CPZH data set. Because this data set also relates to parking facilities, we reuse the datex vocabulary. We searched the corresponding entities on LinkedGeoData to raise the CPBDE data stream to the 5-star open data level. The last triple in Table 10 contains a link to the Web page for the parking facilities which originally is not contained in the data set. We add this in the corresponding connector to increase the information quality of the output stream. Table 10 shows the triples which TripleWave creates with our mapping.

Subject	Predicate	Object
dex:1	dtx:parkingName	bahnhof.txt
dex:1	dtx:parkingNumberOfSpaces	114
dex:1	dtx:numberOfVacantParkingSpaces	039
dex:1	dtx:dateTime	28.06.2017 16:14
dex:1	rdf:type	ldg:ontology/Parking
dex:1	owl:sameAs	ldg: geometry/node596726810
dex:1	dtx:urlLinkAddress	http://bcp-bonn.de/bahnhofgarage/

Table 10: CPBZH Mapping

[46] http://vocab.datex.org/terms/# (accessed 28.6.2017)

[47] http://linkedgeodata.org/page/ontology/Parking (accessed 28.6.2017)

[48] http://www.openstreetmap.org/ (accessed 28.6.2017)

4.3.4 ZR Mapping

The ZR data set is continuously updated and so frequently contains new data. It is a suitable candidate for being published as RDF stream. We have not found a vocabulary which describes bicycle rentals. So we used other vocabularies to describe its content. The only attribute we could not find in an existing vocabulary is the number of free bikes. So we used a custom vocabulary. Unfortunately, only four of the eight rental locations are marked on OpenStreetMap, so half of the records cannot be linked to LinkedGeoData entities. Because the original data set does not contain a time stamp, we add one in the corresponding connector using the number of milliseconds since 1.1.1970 00:00:00 UTC. Table 11 shows the triples which are the result of the mapping TripleWave uses.

Subject	Predicate	Object
dex:6	rdfs:label	Züri rollt Enge
dex:6	ex:numberOfFreeBikes	14
dex:6	purl:date	1498661139479
dex:6	rdf:type	ldg:ontology/BicycleRental
dex:6	owl:sameAs	ldg:geometry/node504484391
dex:6	geo:long	8.53245629597557
dex:6	geo:lat	47.3646919355276

Table 11: ZR Mapping

4.3.5 CPMDE Mapping

The CPMDE data contains data about parking facilities, so we use the vocabulary mentioned above, i.e. datex. We link the records of the data set to LinkedGeoData entities, so we are able to raise the CPMDE data stream to the 5-star open data level. The original data set does not contain time stamps for the records, so we add them in the corresponding connector using UNIX time, i.e. the number of milliseconds since 1.1.1970 00:00:00 UTC.

Subject	Predicate	Object
dex:Parkplatz Mühlenstraße	dtx:parkingName	Parkplatz Mühlenstraße
dex:Parkplatz Mühlenstraße	dtx:address	Moerser Benden
dex:Parkplatz Mühlenstraße	dtx:parkingNumberOfSpaces	999
dex:Parkplatz Mühlenstraße	dtx:numberOfVacantParking-Spaces	789
dex:Parkplatz Mühlenstraße	dtx:dateTime	1498662497395
dex:Parkplatz Mühlenstraße	rdf:type	ldg:ontology/Parking
dex:Parkplatz Mühlenstraße	owl:sameAs	ldg:geometry/way4440755
dex:Parkplatz Mühlenstraße	geo:long	6.62589
dex:Parkplatz Mühlenstraße	geo:lat	51.4553

Table 12: CPMDE Mapping

4.4 Implementing the use case

We present here the components we finally run on the University Server. As not all the implemented components are used, we show in Figure 7 the used components. The final cluster fetches, transforms and streams out the following five data sets: CPZH, ZR, TCZH, CPBDE and CBMDE. We do not connect to already existing WebSocket streams or SSE streams. For the five data sets the corresponding mappings are used. We run for each data set one TripleWave instance for the transformation. As the

incoming data streams are not that large, one TripleWave instance is enough for each data set. We make the resulting RDF streams available over WebSockets and the EventSource interface using the WebSocketConsumer and KafkaSSE components.

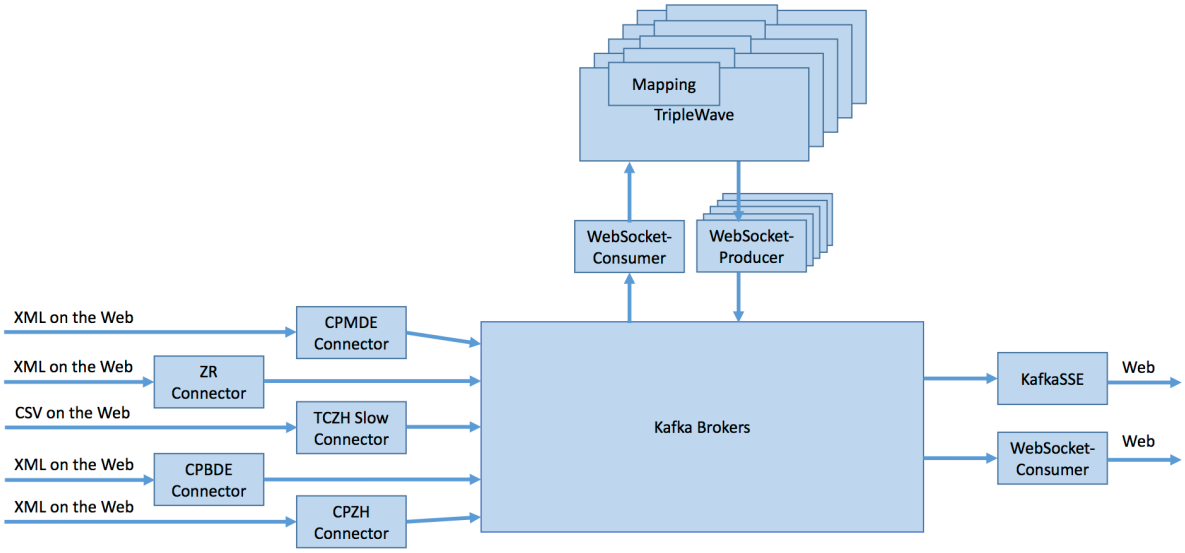

Figure 7: Components run on the University server

There are some components which are missing in Figure 7 in contrast to Figure 6: KafkaThrift, KTClient, SMNRCH Connector, TCZH Fast Connector, WebSocketProducerBig, SSEProducer, WebSocketClient and the EventSource component. The evaluation results of the KafkaThrift component were not as good as the evaluation results of JSON-LD and thus we decided to not run it on the server (see Section 5.4). We implement the SMNRCH Connector to show how such a data set can be fetched. But we did not create a mapping for this data set due to lack of time. The TCZH Fast Connector and WebSocketProducerBig belong together and so if we do not run one component the other one is also not used. These two components are a guideline for processing data streams with high throughput. In Appendix B we give an example of how the two components can be run and in Section 5.2 we show that our cluster scales in combination with them. But as in our project the resulting RDF output streams are published over the Web, we decided to use the TCZH Slow Connector on the server, because at the end, clients must be able to consume the output stream using a simple WebSocket or SSE/EventSource connection. This means that the final output stream must not have too high throughput as the clients could not consume it over the Web. The SSEProducer component is not used because we do not connect to already existing SSE/EventSource streams. Finally, the WebSocketClient and EventSource components are simple client application examples and must be run on client side.

5 Evaluation

In this chapter we present the evaluation results of our prototype. We ran the evaluation on a MacBook Pro (Retina, 13-inch, Early 2015) which has an Intel Core i5-5257U (2.7 GHz, dual-core) processor and 8GB 1867 MHz DDR3 RAM. We present the results of four different tests. In Section 5.1 we compare the latency of the whole prototype (Kafka connected with TripleWave) to the latency of only TripleWave. In Section 5.2 we test the scalability of our system connecting different numbers of TripleWave instances to a data stream. In Section 5.3 we present the time performance of different parts of our implemented cluster. In Section 5.4 we test the size of the data packages in different formats.

5.1 Latency

In this section we present our results of evaluating the latency of different setups of our prototype and comparing it with TripleWave.

For all of the five tests we restarted the Mac Book. No unnecessary programs were run. With these arrangements we want to ensure that all tests have the same preconditions.

In Figure 8 we illustrate the setup of our prototype for this evaluation. The DataSetReader10x component reads a data set which contains 3000 records and sends each of them to Kafka. This data set is processed ten times, for a total of 30'000 records. We have implemented a pause of 20ms between each record, because we do not want to overload the system as we measure the latency and not the throughput. With this approach we can ensure that there are no data bursts. After the Kafka brokers receive the records, data are forwarded to a TripleWave instance which transforms the data using a simple mapping. The resulting RDF stream is again forwarded to Kafka. At the end a client application consumes the RDF stream. We measure the time which a record needs to pass the whole cluster, starting at the DataSetReader10x component and ending at the consumption client.

We repeat this test for four different setups. In Figure 8, KT stands for the connection type between Kafka and TripleWave and KC stands for the connection type between Kafka and the client application. If the connection of KT or KC uses SSE/EventSource, then we run the KafkaSSE component. If the connection of KT or KC uses WebSockets, then we run the WebSocketConsumer component. We tested all four combinations of KT and KC and the results are shown in Table 13 and in Figure 10.

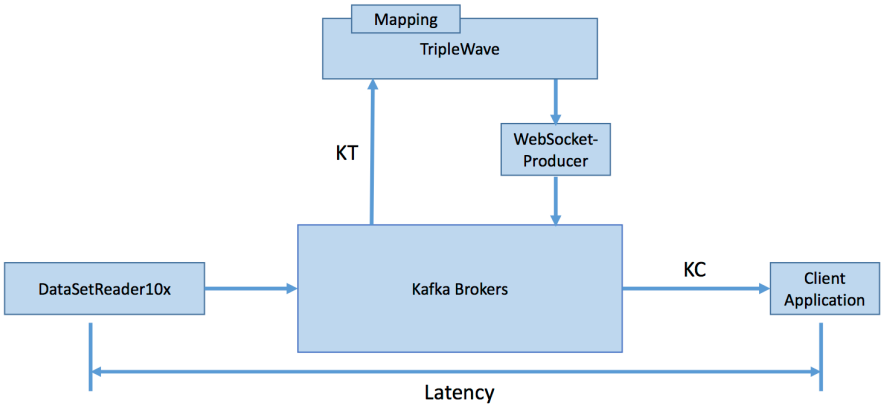

Figure 8: Setup of our cluster for latency test

We measure the latency of TripleWave with the setup illustrated in Figure 9. We implement a DataSetReader10x component in TripleWave and send the RDF output stream to a consumption client application over WebSockets. We measure the time a record needs to pass TripleWave including the DataSetReader10x component and receiving the client application. The results are shown in Table 13 and in Figure 10.

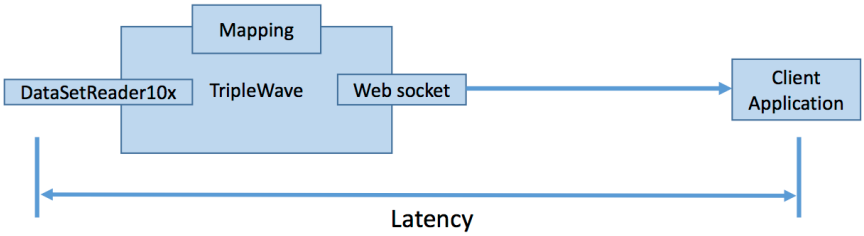

Figure 9: Setup of TripleWave for latency test

Table 13 shows the results of the latency tests. Each column refers to one specific setup. For example, SSE-WS means that the first connection (KT) uses SSE/EventSource and the second connection (KC) uses a WebSocket. The last column contains the latency of TripleWave (TW). Each row (except the last two) stands for one of the ten rounds. In each round there are computed 3'000 latency values, one value for each record of the test data set. The value in one field in Table 13 stands for the average latency in milliseconds for the 3'000 values computed in a certain round. The second last row contains the average latency for all the ten rounds. As the first round for all the setups has a significantly higher latency, we present in the last row the average latency for the rounds 2-10.

	SSE-WS	WS-WS	SSE-SSE	WS-SSE	TW
ROUND 1	5.86ms	5.84ms	6.06ms	5.75ms	0.80ms
ROUND 2	5.14ms	5.04ms	5.10ms	4.88ms	0.75ms
ROUND 3	4.99ms	4.81ms	4.87ms	4.66ms	0.76ms
ROUND 4	4.83ms	4.63ms	4.83ms	4.61ms	0.73ms
ROUND 5	4.84ms	4.59ms	4.72ms	4.60ms	0.75ms
ROUND 6	4.81ms	4.70ms	4.84ms	4.56ms	0.71ms
ROUND 7	4.90ms	4.67ms	4.80ms	4.57ms	0.74ms
ROUND 8	4.93ms	4.71ms	4.77ms	4.56ms	0.72ms
ROUND 9	4.90ms	4.59ms	4.74ms	4.53ms	0.73ms
ROUND 10	4.85ms	4.65ms	4.75ms	4.56ms	0.71ms
AVG (1-10)	5.01ms	4.82ms	4.95ms	4.73ms	0.74ms
AVG (2-10)	4.91ms	4.71ms	4.82ms	4.61ms	0.73ms

Table 13: Results of latency tests

As we can see in Table 13, the average latency of TripleWave is less than one millisecond. But the time stamps we use for these tests contain milliseconds as smallest unit. The results cannot be totally precise, because they are more precise as the used time stamps. But for our comparison of the latency between TW and the cluster containing Kafka and TW the quality of the results suffices.

Figure 10 visualizes the presented values of Table 13. The result for each setup in round one is always notable higher than the results for the other nine rounds. This is because at the beginning Kafka must establish several mechanisms what takes time and increases the latency. After the first round, the average values remain stable. We can draw two important conclusions out of these results. First, we can observe that TripleWave is responsible for about 20% of the latency of the whole cluster. Second, we can observe that there are no remarkable latency differences between using WebSockets or SSE/EventSource.

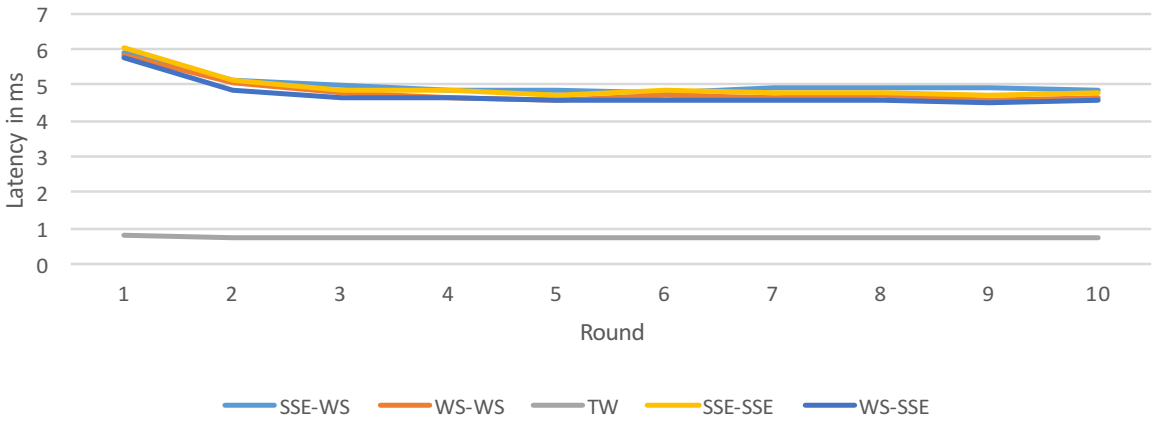

Figure 10: Illustration of latency test results

5.2 Scalability and throughput

In this section, we present the throughput results of our cluster. We measured the throughput for the cluster with one TripleWave instance, two TripleWave instances and three TripleWave instances connected. For each of the three setups we restarted the Mac Book to ensure that each test execution has the same preconditions.

The first setup which includes one TripleWave instance is constructed as follows: We use the TCZH Fast Connector which we slightly modified for this test. It fetches the corresponding data set but forwards only 50'000 records. We do not want the WebSocket connection from Kafka to TripleWave to disconnect because of a buffer overflow (with 50'000 records this does not happen). The 50'000 records are sent in one batch from the TCZH Fast Connector to the Kafka brokers, so that afterwards no more processing power is needed for the Kafka input. The records are then forwarded to the TripleWave instance over a WebSocket connection (using the WebSocketConsumer component). TripleWave transforms the records according to the TCZH.r2rml mapping. Then the records are sent back to Kafka with the help of the WebSocketProducerBig component. In the WebSocketProducerBig component for each incoming record a time stamp is logged which we can use to calculate the records per second processing rate.

For the setup which includes two TripleWave instances, the TCZH Fast Connector forwards 50'000 records to each TripleWave instance (using two different topics). The TCZH Fast Connector parses 100'000 records to JSON at once so that the two batches à 50'000 records can be sent to Kafka as simultaneously as possible. With this approach we achieve that the two TripleWave instances start working at mostly the same time. The two TripleWave instances send the output to two different WebSocketProducerBig components which each log time stamps for processed records.

For the setup which includes three TripleWave instances, we modified the TCZH Fast Connector so that it converts 150'000 records at once and then forwards 50'000 records to three different topics as simultaneously as possible. Each of the three TripleWave instances sends the output data to a different WebSocketProducerBig component which logs time stamps.

As mentioned above, we choose 50'000 records per TripleWave instance (and not more) because otherwise the WebSocket connection between Kafka and TripleWave disconnects after certain time, since the buffer overflows. This means that Kafka can handle higher throughput than the WebSockets or TripleWave. We do not know, however, where exactly the bottle neck is, but it is an interesting future investigation.

In Section 5.1 we showed that at the beginning the latency is higher because our cluster must first initialize several processes. Because of that fact we run each of the setups twice, without restarting the cluster. For each of the test executions we run a console-consumer for each of the needed topics to see

when the processing for each topic is done. A console-consumer is a standard component which is included in the download package of Kafka[49].

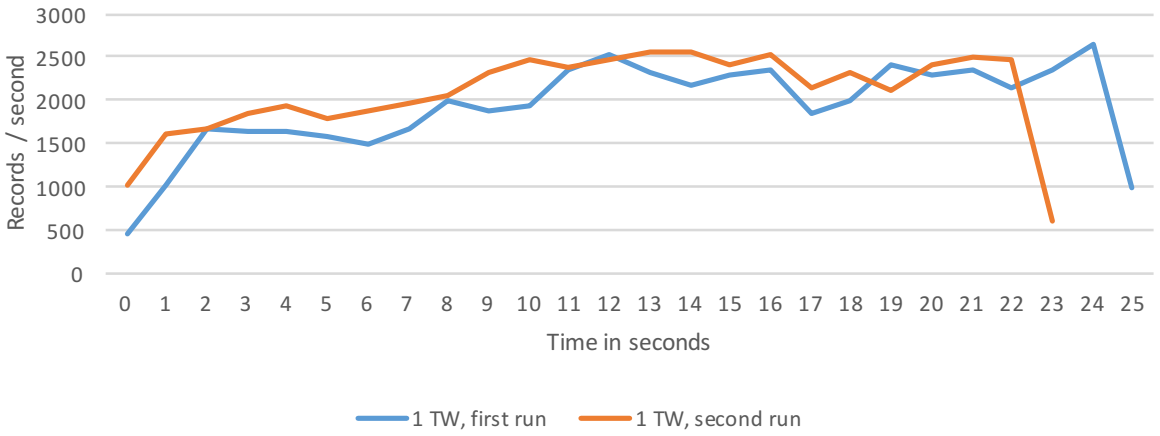

Figure 11: Throughput with one TripleWave (TW) instance

Figure 11 shows the throughput in records per second over the time which was needed to process the 50'000 records. *1 TW, first run* means that the results belong to the setup with one TripleWave instance for the first test execution. *1 TW, second run* means that the results belong to the same setup but for the second test execution. We can see that for the second test execution our cluster does need less time to reach a stable level of throughput. For both setups the maximum throughput is around 2'500 records per second.

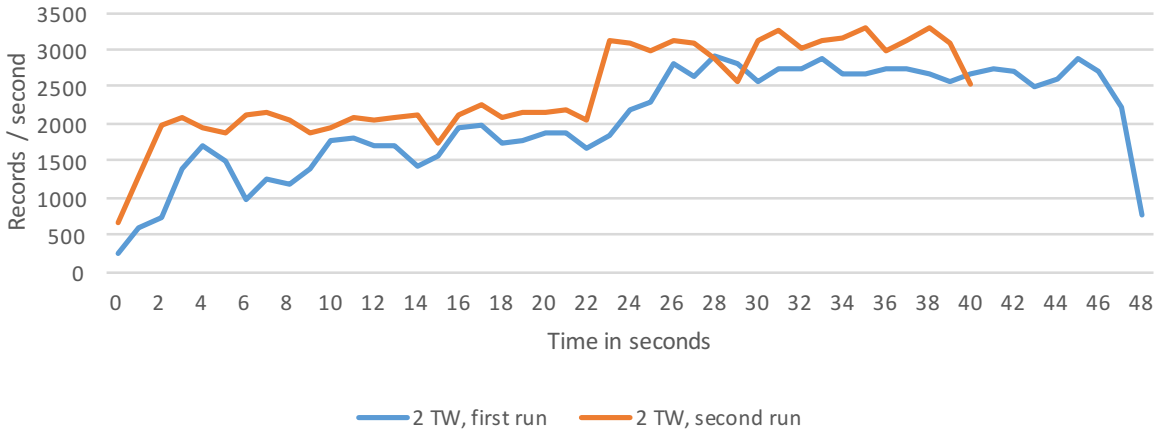

Figure 12: Throughput with two TripleWave (TW) instances

Figure 12 shows the measured throughput for the setup with two TripleWave instances connected to Kafka which process overall 100'000 records. The annotation is similar to Figure 11. *2 TW* means that the setup contains two TripleWave instances and again a first and a second test are executed. We can see that the maximum throughput is higher than for the setup with only one TripleWave instance. In the *1 TW* setup the maximum throughput is around 2'500 records per second. In the *2 TW* setup the maximum throughput reaches 3'330 records per second. Again we observe that in the second test execution

[49] https://kafka.apache.org/quickstart (accessed 10.7.2017)

less time is needed to reach a stable level of throughput. What we also can observe is that there are two phases in the timeline where the throughput is stable on two different levels.

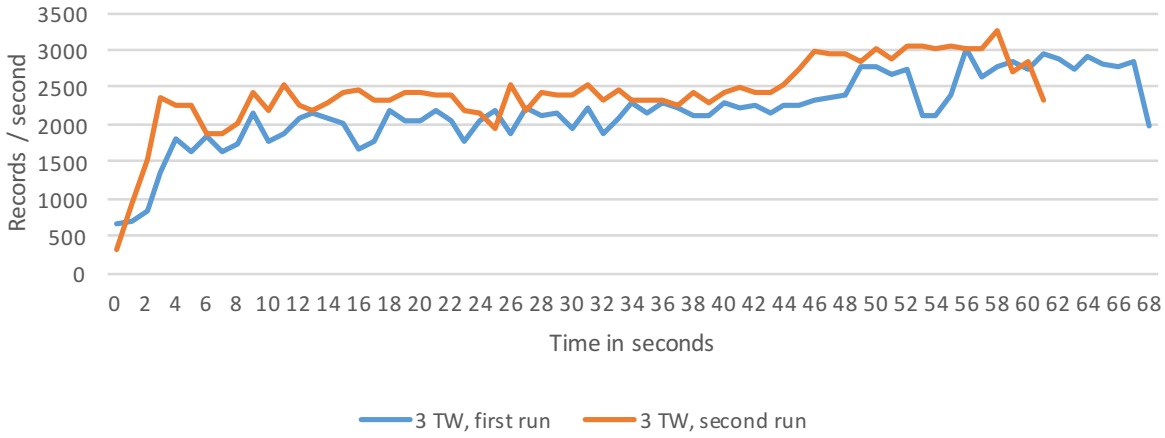

Figure 13: Throughput with three TripleWave (TW) instances

Figure 13 shows the results for the setup with three TripleWave instances connected to Kafka. In both runs are processed 150'000 records. The maximum throughput for *3 TW, first run* is slightly higher than the test results *in 2 TW, first run.* But the maximum throughput for the *3 TW, second run* is lower than for the *2 TW, second run*. Again we see the two phases with stable throughput of different levels and that in the second run a stable level of throughput is reached faster.

We cannot explain the two different levels of stable throughput with certainty. But we hypothesise that there are some processes going on in Kafka which require at the beginning (where the 50'000 records per TripleWave instance are sent to Kafka) more process power than towards the end of a test execution. And so, towards the end the TripleWave instances can use more process power. Because of these two different levels of throughput we calculate two different throughput averages per test execution. Since all test executions show the same behaviour we can compare the averages between the different setups.

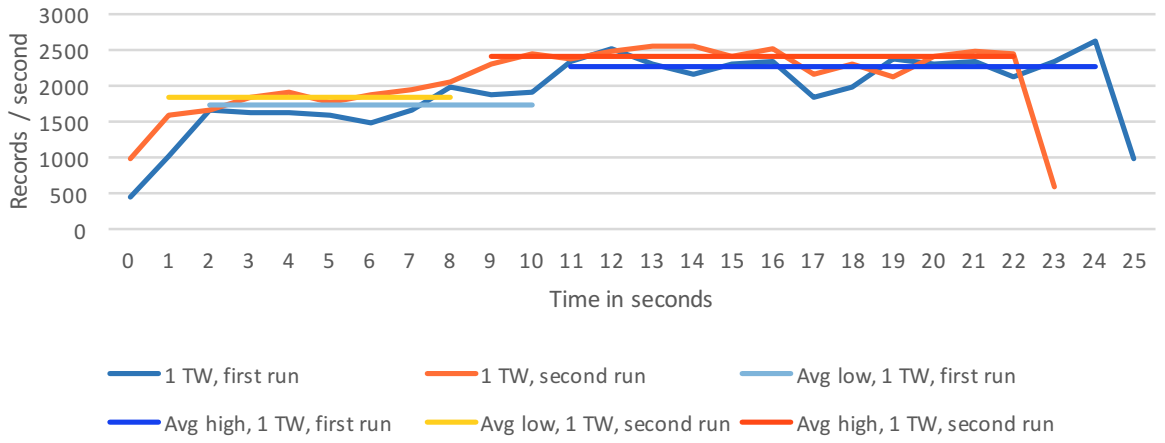

Figure 14: Throughput with one TripleWave (TW) instance with averages

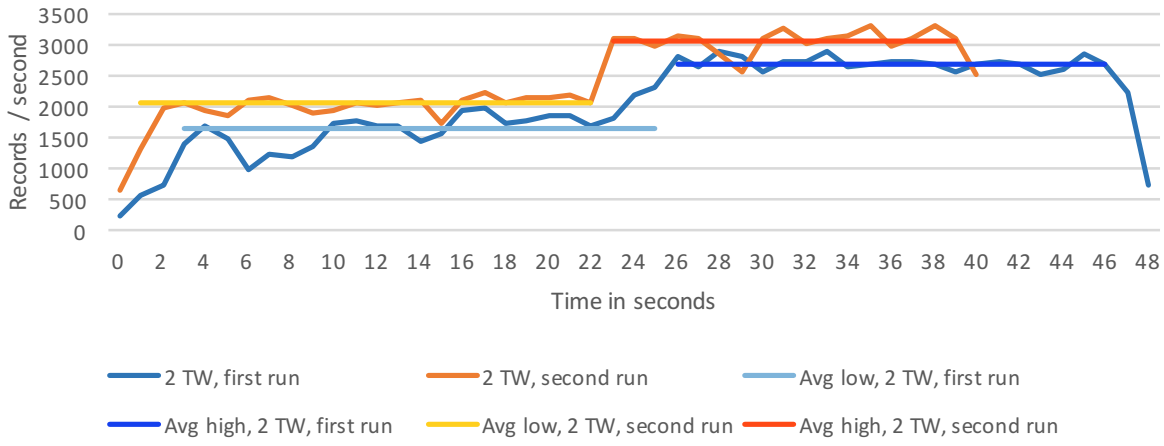

Figure 15: Throughput with two TripleWave (TW) instances with averages

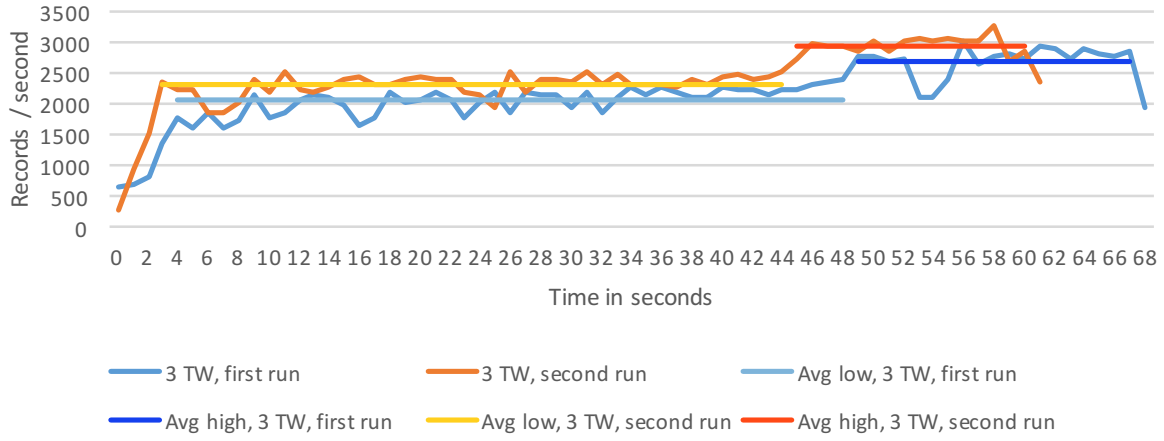

Figure 16: Throughput with three TripleWave (TW) instances with averages

In Figures 14-16 we show the throughput rates of each of the three setups with the different averages illustrated. Each figure contains the throughput rates for the two test executions and four different averages. For the first and the second run of a setup the average throughput of the lower level and the higher level of stable throughput is illustrated. We leave away some of the first values for the average computation because there the prototype has an initialization phase and the throughput is not stable. We also do not consider the last few values, since not all TripleWave instances finish exactly at the same time. We see that in all three setups the records are processed faster in the second run than in the first run. The averages are higher and there is less time required for the initialization phase, thus the average lines begin and end earlier in the timeline. We report the illustrated values of Figures 11-16 in Appendix A.

	1 TW INSTANCE (RECORDS/SEC)	2 TW INSTANCES (RECORDS/SEC)	3 TW INSTANCES (RECORDS/SEC)
AVERAGE LOW, FIRST RUN	1756	1674	2064
AVERAGE HIGH, FIRST RUN	2289	2712	2714
AVERAGE LOW, SECOND RUN	1842	2053	2321
AVERAGE HIGH, SECOND RUN	2403	3083	2963

Table 14: Throughput averages

Table 14 shows the averages for the low and high throughput levels for each setup and each test execution. We expect that the setup with two TripleWave instances generates the highest throughput because the Mac Book processor contains two cores. As we see this is only the case for the last row in the table. Thus the two TripleWave instances are not distributed most efficiently between the two cores.

The main statement we want to show with this evaluation is that our cluster scales. In every row the setup with three TripleWave instances processes more records as the setup with one TripleWave instance. Also the setup with two TripleWave instances processes more records per second than the setup with one TripleWave instance except for the first row. Especially in the high throughput phase, the two TripleWave setup processes significantly more records per second.

5.3 Time performance of Kafka and TripleWave

In this section we present the time performance analysis of our prototype. We measure the time at several spots to find out how long each part of our prototype needs to process the data. In Section 5.1 we found out that there are only small latency differences using different connection types. So we decided to set up our prototype with WebSocketConsumer components as we can configure the WebSocketConsumer easier than KafkaSSE, and add a time stamp writing mechanism.

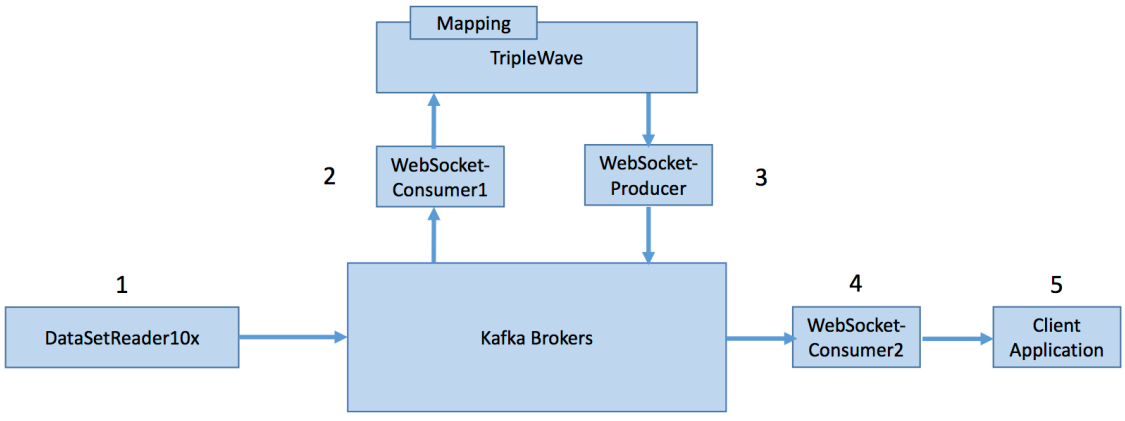

Figure 17: Setup of time performance evaluation

In Figure 17 we illustrate the setup for this test. At five different spots we log time stamps, so we are able to define the processing time of four different passages. The numbers in Figure 17 show where we

log the time stamps. The first time stamp is logged in the DataSetReader10x before the records are sent to the Kafka brokers. The second time stamp is logged in the WebSocketConsumer1 component before forwarding the data to TripleWave. The third time stamp is logged in the WebSocketProducer component before sending the records back to Kafka. The fourth time stamp is again logged in a WebSocketConsumer component before the data are forwarded. The last time stamp is logged after the data are received in the client application.

	1-2	2-3	3-4	4-5	OVERALL
ROUND 1	2.56ms	1.52ms	2.32ms	0.64ms	7.04ms
ROUND 2	2.04ms	1.26ms	1.64ms	0.52ms	5.46ms
ROUND 3	1.91ms	1.22ms	1.58ms	0.52ms	5.23ms
ROUND 4	1.92ms	1.21ms	1.57ms	0.47ms	5.17ms
ROUND 5	1.89ms	1.18ms	1.53ms	0.46ms	5.06ms
ROUND 6	1.95ms	1.21ms	1.45ms	0.46ms	5.07ms
ROUND 7	2.01ms	1.23ms	1.49ms	0.46ms	5.19ms
ROUND 8	1.99ms	1.20ms	1.41ms	0.44ms	5.04ms
ROUND 9	1.99ms	1.21ms	1.45ms	0.43ms	5.08ms
ROUND 10	2.00ms	1.18ms	1.45ms	0.44ms	5.07ms
AVG (1-10)	2.03ms	1.24ms	1.59ms	0.48ms	5.34ms
AVG (2-10)	1.97ms	1.21ms	1.51ms	0.46ms	5.15ms

Table 15: Results of time performance test

In Table 15 we present the test results. In column *1-2* we show the time which is needed for a data record to be processed from the DataSetReader10x component to the WebSocketConsumer1 component. In column *2-3* we show the processing time which is needed between the WebSocketConsumer1 component and the WebSocketProducer component, and so on. The last column shows the overall time needed for the whole cluster. The second last row contains the average processing time for all the ten rounds. As the processing time for the first round is significantly higher, we present in the last row the average processing time for all rounds excluding the first one.

We expect the overall time to be the same as in Table 13 column *WS-WS*, because the setup is quite similar. There are only two differences between the two setups: first, for the time performance test, we log time stamps at not only two spots but at five different spots. Second, we use two WebSocketConsumer components as it is easier to implement two timestamp logging mechanisms in two different components. Thus slightly more processing is required in this setup compared to Section 5.1. We explain the small time difference between the overall time and Table 13 column *WS-WS* with this additional processing effort.

We expect the processing times in column *1-2* and column *3-4* to be the same, as both passages contain a Producer, the Kafka brokers and a Consumer. But there is a difference about half a millisecond. We explain this difference with the different data objects processed on this two passages. Between the DataSetReader10x component and the WebSocketConsumer1 component the non-RDF JSON objects are processed which are considerably bigger than the transformed JSON-LD objects which are processed between the WebSocketProducer component and the WebSocketConsumer2 component. This is because a lot of attributes of the original JSON object are ignored in the corresponding R2RML mapping.

As mentioned above, the test results are more precise than the time stamps used for the evaluation. Thus a certain variance from the real processing time is possible. But nevertheless the presented results give an idea of the processing times needed for the different passages.

5.4 Data exchange comparison

In this section, we present the results of measuring the package sizes of the content of an RDF output stream. We measure the package sizes of RDF output stream elements for different formats and for different connection types. We measure the package size with a framework called Wireshark[50].

On the one hand we want to measure the package size of the JSON-LD objects which are the output of TripleWave and on the other hand we want to measure the package size when we apply the RDF Binary scheme using Apache Thrift to transmit the data. The JSON-LD objects are sent using WebSockets and SSE/EventSource. Thus we can compare the package size difference using the two connection mechanisms. We run the Apache Thrift transmission with our KafkaThrift and KTClient components (see Figure 6, Section 4). We execute two Apache Thrift tests. We use in the first test the TBufferedTransport in combination with the TCompactProtocol and in the second test the TBufferedTransport in combination with the TBinaryProtocol. So we can compare the package sizes for these two different Protocols.

Connection Type	Individual packages (in bytes)	Final result (in bytes)
SSE – JSON-LD	116 + 453	569
WebSockets – JSON-LD	374	374
TCompactProtocol – Thrift	192 + 237 + 224 + 230	883
TBinaryProtocol – Thrift	231 + 285 + 272 + 278	1066

Table 16: Package sizes

We present the four results in Table 16. The first column contains the four different connection types. The second column contains the sizes of the individual packages in bytes. A Server Sent Event contains an ID object and a message object. The 116 bytes are the ID object and the 453 bytes stand for the message object which contains a JSON-LD object. Using WebSockets, there is sent only one frame which contains the JSON-LD object. When we use our KafkaThrift and KTClient components to transmit the JSON-LD object, it had to be converted to NQuads. There were four Quads necessary to describe the corresponding JSON-LD object thus in row four and five there are four package sizes. The last column sums up the individual package sizes.

The size of the records of our test data set differ to a small degree from record to record. This is because the records contain data about user names and other things which require different numbers of bytes to be encoded. To be able to directly compare the test results, we present in Table 16 the package sizes for the same record of the test data set.

We can observe that using WebSockets generates the smallest package size followed by SSE. This makes sense because of several reason. First a Server Sent Event contains not only the message object (which contains the actual JSON-LD) but also an ID object. Thus the whole package must be larger. Second, a SSE/EventSource connection works over HTTP (Hickson 2015), but a WebSocket works over TCP/IP[51]. It is therefore expected that SSE/EventSource introduces additional overhead.

The two connections using Apache Thrift need significantly more bytes than the SSE and WebSockets connection. This is because the RDF Binary scheme is not efficient for our output. What is written in one JSON-LD object must be converted to four NQuads to be transmitted. This means that four packages must be sent. We can additionally confirm with this test that the TCompactProtocol needs less bytes to transmit the same data than the TBinaryProtocol.

We only evaluated Apache Thrift in combination with the RDF Binary scheme. The RDF Binary scheme compresses triples but the JSON-LD output contains RDF graphs. It would be interesting to write a

[50] https://www.wireshark.org/ (accessed 5.7.2017)

[51] https://en.wikipedia.org/wiki/WebSocket (accessed 5.7.2017)

scheme which compresses RDF graphs and repeat this evaluation with the created scheme. Apache Thrift may start to pay off in combination with a scheme that is optimized for the output of TripleWave.

6 Conclusion

Our study deals with the lack of guidelines and prototypes which show how open data sets can be published as Linked Data streams. So we developed a system which fetches, transforms and publishes several open data sets as RDF streams. We wrote down our experience and challenges and thus this work can be seen as guideline for future projects.

We examined Swiss, German and Austrian OGD portals and we made a survey of potential streaming data sets. The focus of the examination was on the Swiss OGD portals. We examined the OGD portals of Zurich, Switzerland, SBB and Swiss Public Transport. In addition, we extended our survey with some data sets from German and Austrian OGD portals.

Our prototype is a combination of Kafka and TripleWave. We use it to increase the number of Linked Open Data streams on the Web, contributing to the development of the Linked Data Web.

The prototype consists of several components which manage individual tasks. We split our components into three groups: input components, output components and R2RML mapping components.

We developed two types of input components. The first type of components are important parts of the connection between Kafka and TripleWave. The other kind of components, called connectors, fetch, clean, enrich and forward several open data sets to Kafka. We have written connectors for different types of data sets. These connectors serve as example connectors for accessing different types of data sets. As every data set has its own characteristics, e.g. data format or protocol, connectors may require to be adapted if used for other data sets.

The output components have two different usages. First, they are used for the connection between Kafka and TripleWave. Second, they publish the RDF streams in the Web. We used three different types of output connections: WebSocket, SSE and Apache Thrift.

The R2RML mapping components are required by TripleWave to transform the non-RDF input streams into RDF output streams. For every of the five data sets we considered, we wrote an individual mapping. All five input data sets are classified as 3-star open data according to the Tim Berners-Lee 5-star open data scheme. Our goal was to increase the number of stars: we used links to entities on LinkedGeoData for four of the five data sets, thus the RDF output streams are 5-star open data. We also enriched some of the data sets with additional information. e.g. time stamps or URLs to Web sites of the entities.

Our prototype is modular and scalable. We used Kafka and its topics to manage the different data streams. Because we wrote individual components which are connected to Kafka and TripleWave our cluster can easily be extended. New data streams can be connected at runtime, thus no client applications which are already connected to our cluster are affected. To fetch and transform a new open data set and publish it as RDF stream, users only have to do few steps: adapt a connector, write an R2RML mapping, create new Kafka topics and connect the adapted connector and a TripleWave instance to the cluster. The new stream can be consumed by client applications. Our prototype transforms data sets which are available through pull-based mechanisms into streams which can be consumed through push-based protocols like SSE and WebSockets.

After we developed our prototype, we ran different tests to evaluate it. We measured the latency in different setups and determined the processing times of its different parts. We showed that the cluster is scalable by measuring the throughput for different numbers of TripleWave instances connected to Kafka. Finally, we compared the package sizes of output stream objects for different push-based mechanisms.

There are some limitations in our work. We examined all data sets from the OGD portal of Zurich, from the OpenTransportData portal and from the SBB portal. But as the OGD portals of Switzerland, Germany and Austria contain thousands of data sets, we were not able to examine all of them. Thus our survey is non-exhaustive.

There are some limitations corresponding to our mappings. Our TCZH mapping contains a custom vocabulary because at the best of our knowledge, there is no existing vocabulary for this data set. Our ZH

mapping links entities from LinkedGeoData to the entities of the data set. But LinkedGeoData contains only four of eight required entities and thus we were only able to link half of the entities of the data set. For the CPZH mapping the same problem exists but there only two out of 37 entities missing. So we were able to link 35 of 37 parking facilities of the data set to LinkedGeoData.

Our evaluation also includes some limitations. The throughput evaluation has shown that our prototype scales. We also saw that Kafka can manage higher throughput than TripleWave in combination with WebSockets. But with our evaluation we did not show where exactly the bottleneck is. Another limitation of our evaluations is that they are done on localhost. This means that no network connection was included in the tests. So our latency test results do not consider the advantages or disadvantages of WebSockets or SSE concerning a network connection.

We have learned a lot of things during this thesis. First of all, we learned about the underlying technologies, namely Node.js, TripleWave, Apache Kafka and Apache Thrift. As TripleWave is written in JavaScript, the output streams are published on the Web and therefore client applications are likely to be written in JavaScript we decided to write our components also in JavaScript. We did not regret this decision because there exist a lot of good packages for Node.js which were useful for our work. On the one hand Node.js packages enabled to interact with Kafka which is written in Scala and Java[52]. On the other hand, we found a lot of helpful packages for various tasks, e.g. converting CSV and XML to JSON or to set up WebSockets.

We noticed that Kafka is a simply usable framework to manage data streams. The combination of TripleWave and Kafka was a good decision as the resulting cluster can easily be extended and clearly manages the different streams. The use of Apache Thrift in combination with the RDF Binary scheme was not a good choice, as the RDF Binary scheme is not efficient for our output format. The data can be transmitted in smaller packages when we use JSON-LD and SSE or WebSockets. If there is a Thrift scheme which is more optimized for the TripleWave output, Apache Thrift may start to pay off as an alternative to serialize the RDF streams.

An unexpected finding is that there are quite a few real-time data sets on the OGD portals in relation to the total number of data sets. The most real-time data sets contain information about car parks and the number of free parking spaces. It seems that such data are easy to collect and to publish on the OGD portals. We also expected to find more real-time public transportation data. But the public transportation data sets which can be downloaded only contain the data of the previous day.

There exist a lot of vocabularies for Linked Data. But it is hard to find the suitable vocabularies for a given data set. The Linked Open Vocabularies Web site[53] helped us in this direction, but we register the need of more tools to enable the identification and reuse of existing vocabularies. Concerning the linking of the selected data sets LinkedGeoData is a good option. LinkedGeoData benefits from OpenStreetMap, which can be edited by everyone. Thus the number of entities on OpenStreetMap and LinkedGeoData increases over time.

For future work it would be interesting to use our prototype with sensor data from Internet of Things. A prototype can be developed to consume the data of a sensor network, to transform them into Linked Data and then use SPARQL (or a continuous extension) to query the data streams. Because the data streams are in RDF, the evaluation process can include data from related data sources distributed over the Web. This would be a step further towards integrating the topic Internet of Things into the Linked Data Web.

[52] https://en.wikipedia.org/wiki/Apache_Kafka (accessed 12.7.2017)

[53] http://lov.okfn.org/dataset/lov/ (accessed 12.7.2017)

7 Bibliography

Barbieri, DF & Della Valle, E 2010, 'A Proposal for Publishing Data Streams as Linked Data', *Proceedings of the Linked Data on the Web Workshop (LDOW2010),* Raleigh, North Carolina.

Berners-Lee, T 2006, *Linked Data,* viewed 5 June 2017, <https://www.w3.org/DesignIssues/LinkedData.html>.

Berners-Lee, T, Fielding, R, & Masinter, L 2005, *Uniform Resource Identifier (URI): Generic Syntax. Request for Comments: 3968,* viewed 23 July 2017, <https://tools.ietf.org/html/rfc3986>.

Bizer, C, Heath, T, & Berners-Lee, T 2009, 'Linked Data - The Story So Far', *International Journal on Semantic Web and information Systems*, vol. 5, no. 3, pp. 1-22.

Boyle, D, Yates, D, & Yeatman, E 2013, 'Urban Sensor Data Streams: London 2013', *IEEE Internet Computing*, vol. 17, no. 6, pp. 12-20.

Brickley, D, & Guha, R 2014, *RDF Vocabulary Description Language 1.0: RDF Schema - W3C Recommendation*, viewed 23 July 2017, <https://www.w3.org/TR/rdf-schema/>.

Cyganiak, R, Wood, D, & Lanthaler, M 2014, *RDF 1.1 Concepts and Abstract Syntax - W3C Recommendation*, viewed 11 July 2017, <https://www.w3.org/TR/rdf11-concepts/>.

Fette, I, & Melnikov, A 2011, *The WebSocket Protocol. Request for Comments: 6455*, viewed 23 July 2017, <https://tools.ietf.org/html/rfc6455>.

Fielding, R, Gettys, J, Mogul, J, Frystyk, H, Masinter, L, Leach, P, & Berners-Lee, T 1999, *Hyptertext Transfer Protocol -- HTTP/1.1. Request for Comments: 2616,* viewed 23 July 2017, <https://tools.ietf.org/html/rfc2616>.

Hausenblas, M 2015, *5 ★ Open Data,* viewed 3 Mai 2017, <http://5stardata.info/en/>.

Heyvaert, P, Taelman, R, Verborgh, R, & Mannens, E 2016, 'Linked Sensor Data Generation using Queryable RML Mappings', *Proceedings of the 2016 International Semantic Web Conference Posters & Demonstrations Track.*

Hickson, I 2015, *Server-Sent Events - W3C Recommendation*, viewed 11 July 2017, <https://www.w3.org/TR/eventsource/>.

Kobilarov, G, Scott, T, Raimond, Y, Oliver, S, Sizemore, C, Smethurst, M, Bizer, C, Lee, R 2009, 'Media Meets Semantic Web - How the BBC Uses DBpedia and Linked Data to Make Connections', *The Semantic Web: Research and Applications. ESWC 2009,* pp. 723-737.

Kreps, J, Narkhede, N, & Rao, J 2011, 'Kafka: a Distributed Messaging System for Log Processing', *NetDB'11.*

Mauri, A, Calbimonte, J-P, Dell'Aglio, D, Balduini, M, Brambilla, M, Della Valle, E, & Aberer, K 2016., 'TripleWave: Spreading RDF Streams on the Web', *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference*, Kobe, Japan, pp. 140-149.

McGuinness, DL, & van Harmelen, F 2004, *OWL Web Ontology Language - W3C Recommendation*, viewed 23 July 2017, <https://www.w3.org/TR/owl-features/>.

Open Knowledge International 2017a, *What is open?*, viewed 31 Mai 2017, <https://okfn.org/opendata/>.

Open Knowledge International 2017b, *Open Definition 2.1*, viewed 31 Mai 2017, <http://opendefinition.org/od/2.1/en/>.

Open Knowledge International 2017c, *Open Government Data*, viewed 31 Mai 2017, <https://opengovernmentdata.org/>.

Scott, A 2015, *Citymapper executive to governments: 'Open more data so we can improve your cities'*, viewed 31 Mai 2017, <https://theodi.org/news/citymapper-government-open-data-improve-cities>.

Shadbolt, N, O'Hara, K, Berners-Lee, T, Gibbins, N, Glaser, H, Hall, W, & Schreafel, M 2012, 'Linked Open Government Data: Lessons from Data.gov.uk.', *IEEE Intelligent Systems*, vol. 27, no. 3, pp. 16-24.

Slee, M, Agarwal, A, & Kwiatkowski, M 2007, *Thrift: Scalable Cross-Language Services Implementation*, Technical Report, Facebook.

Taelman, R, Heyvaert, P, Verborgh, R, & Mannens, E 2016a, 'Querying Dynamic Datasources with Continuously Mapped Sensor Data'.

Taelman, R, Verborgh, R, Colpaert, P, Mannens, E, & Van de Walle, R 2016b, 'Continuously Updating Query Results over Real-Time Linked Data', *Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web.*

Vineet, J, & Xia, L 2017, 'A Survey of Distributed Message Broker Queues', *eprint arXiv:1704.00411*.

Vrandencic, D, & Krötzsch, M 2014, 'Wikidata: A Free Collaborative Knowledgebase', *Communications of the ACM*, vol. 57, no. 10, pp. 78-85.

Wang, G, Koshy, J, Subramanian, S, Paramasivam, K, Zadeh, M, Narkhede, N, Rao, J, Kreps, J, Stein, J. 2015, 'Building a Replicated Logging System with Apache Kafka', *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1654-1655.

Appendix A: Detailed experimental results

This table contains the values which we present in Section 5.2.

Time in seconds	1 TW, first run (Records per second)	1 TW, second run (Records per second)	2 TW, first run (Records per second)	2 TW, second run (Records per second)	3 TW, first run (Records per second)	3 TW, second run (Records per second)
0	465	1005	243	650	677	304
1	1016	1607	587	1303	716	953
2	1681	1667	739	1996	838	1524
3	1645	1857	1408	2071	1364	2347
4	1633	1933	1690	1946	1794	2241
5	1580	1774	1491	1861	1624	2249
6	1504	1886	971	2125	1852	1873
7	1660	1961	1246	2152	1628	1867
8	1985	2055	1192	2043	1724	2012
9	1884	2315	1377	1887	2141	2414
10	1922	2459	1759	1959	1776	2183
11	2358	2385	1797	2088	1883	2528
12	2540	2483	1688	2048	2072	2246
13	2321	2551	1702	2074	2153	2201
14	2181	2545	1438	2126	2101	2288
15	2302	2405	1570	1726	1998	2414
16	2340	2521	1958	2132	1652	2455
17	1839	2152	1968	2256	1760	2328
18	1989	2326	1728	2071	2204	2334
19	2395	2117	1773	2143	2035	2415
20	2305	2413	1870	2152	2056	2435
21	2358	2502	1876	2199	2185	2392
22	2133	2469	1685	2059	2054	2401
23	2353	612	1840	3126	1772	2204
24	2634		2180	3104	2038	2162
25	977		2304	2974	2178	1956
26			2798	3136	1880	2519
27			2635	3100	2210	2187
28			2904	2877	2133	2416
29			2812	2555	2143	2398
30			2576	3121	1954	2381
31			2730	3262	2214	2530
32			2735	3019	1860	2314
33			2890	3129	2100	2471
34			2662	3169	2279	2326
35			2687	3303	2168	2330
36			2726	2992	2277	2333
37			2739	3128	2211	2264
38			2675	3304	2120	2415
39			2578	3104	2105	2306
40			2674	2530	2295	2444

41			2731		2239	2489
42			2700		2250	2420
43			2514		2169	2428
44			2601		2253	2545
45			2876		2246	2756
46			2698		2337	2988
47			2222		2358	2956
48			757		2392	2935
49					2760	2852
50					2763	3006
51					2688	2874
52					2731	3045
53					2131	3050
54					2104	3028
55					2399	3057
56					3025	3033
57					2642	3014
58					2776	3269
59					2838	2709
60					2741	2843
61					2957	2343
62					2887	
63					2745	
64					2921	
65					2826	
66					2777	
67					2856	
68					1965	

Table 17: Records per second for throughput evaluation

Appendix B: How to run

We give in this section two examples of how to run parts of the cluster. As the mechanisms (fetching, transforming and streaming out) of all five connectors shown in Figure 7 (on page 44) work similar, we give an example for one of them, namely the CPZH Connector. The second example explains how to run the TCZH Fast Connector and thus we show how our cluster can be used for streams with high throughput. As for both examples certain similar things must be done. We describe them before going to the specific examples. The examples below are run on localhost. If this is not the case, the URLs must be adapted.

We use Kafka version 2.11-0.10.2.0 (the newest at the time of the thesis). For each topic that we generate, we use a replication factor of three to guarantee more reliability than with a replication factor of one. We only create one partition for each topic, because we do not need more throughput and the Kafka brokers are not distributed[54].

The second framework which is required is TripleWave[55].

To run TripleWave and the implemented connectors, Node.js is required. For this project we use Node.js version 6.10.2 as TripleWave requires Node.js 6.

Every component needs certain Node.js packages. At the beginning of each file we listed the required packages[56].

How to fetch, transform and stream out the CPZH data set:

- Set up a Kafka cluster with two topics, e.g. cpzh-original and cpzh.
- Run WebSocketConsumer and KafkaSSE. This can be done using the following commands:
    - node pathToFolder/WebSocketConsumer.js
    - node pathToFolder/KafkaSSE.js
- Configure a TripleWave instance and run it. This includes the following steps:
    - In the config.properties file, define some free ports (for the variable ws_port we use for this example the port 4040)
    - In the config.properties file, choose the desired transform_transfomer (wsStream.js or sseStream.js)
    - In the config.properties file, define path and ws_stream_location (in this case we define path=/triplewave and ws_stream_location=/primus; the two variables define the pathname which must be used to connect a WebSocketProducer to TripleWave)
    - In the config.properties file, choose the correct transform_mapping (in this case CPZH.r2rml)
    - In the sseStream.js or wsStream.js file, configure the correct Kafka topic:
        - sseStream.js: const url = 'http://localhost:6917/cpzh-original'; (Line 5)
        - wsStream.js: var url = 'ws://localhost:4041/?topics=cpzh-original'; (Line 3)
    - run TripleWave using the following command:
        - sh pathToFolder/start.sh

[54] For more information of how to run Kafka, see here: https://kafka.apache.org/quickstart (accessed 1.7.2017)

[55] A description of how to download and run it can be found here: http://streamreasoning.github.io/TripleWave/ (accessed 1.7.2017)

[56] To install a package, first install Node.js and then simply use the following command in the command line where packageName is the name of the package to install: *npm install packageName.* If two components need the same package, the package must be installed only once.

- Run a WebSocketProducer to forward the output of TripleWave back to the Kafka cluster. The URL address, pathname and the port must match the configurations of the TripleWave instance:
    - node pathToFolder/WebSocketProducer.js –t=cpzh –u=ws://localhost:4040 –p=/triplewave/primus
- As the whole cluster is now set up, the CPZH Connector can be started:
    - Node pathToFolder/CPZH.js –t=cpzh-original

After these steps are done, clients can connect and consume the resulting RDF stream. We describe in Appendix C how this can be done.

Now we describe how to set up several TripleWave instances for one high throughput data stream illustrated for the TCZH Fast Connector. The TCZH Fast Connector splits the data set into three subsets which each are forwarded to three different Kafka topics. This means that we can connect three TripleWave instances for this data set. The code of the TCZH Fast Connector can easily be changed and the number of subsets can be increased/decreased if desired thus increasing/decreasing the number of connected TripleWave instances. But in the following example, it is shown how to set up the cluster for three TripleWave instances:

- Set up a Kafka cluster with four topics, e.g. tczh-original0, tczh-original1, tczh-original2 and tczh (tczh contains the resulting RDF stream of all three TripleWave instances).
- Run WebSocketConsumer and KafkaSSE. This can be done using the following commands:
    - node pathToFolder/WebSocketConsumer.js
    - node pathToFolder/KafkaSSE.js
- Configure three TripleWave instances and run them. This includes the following steps:
    - In each config.properties file, define some free ports (for the variable ws_port we use for this example 4038, 4039 and 4040)
    - In each config.properties file, choose the desired transform_transfomer (wsStream.js or sseStream.js)
    - In each config.properties file, define path and ws_stream_location (in this case we define path=/triplewave and ws_stream_location=/primus; the two variables define the pathname which must be used to connect a WebSocketProducerBig to TripleWave)
    - In each config.properties file, choose the correct transform_mapping (in this case TCZH.r2rml)
    - In the sseStream.js or wsStream.js file, configure the correct Kafka topic (important: each TripleWave instance must connect to a different subset of the data set; this means tczh-original0, tczh-original1 and tczh-original2 must be used once):
        - sseStream.js: const url = 'http://localhost:6917/tczh-original0'; (Line 5)
        - wsStream.js: var url = 'ws://localhost:4041/?topics=tczh-original0'; (Line 3)
    - run each TripleWave instance using the following command:
        - sh pathToFolder/start.sh
- Run three WebSocketProducerBig to forward the three output streams of TripleWave back to the Kafka cluster. Each output stream is in this example forwarded to the same topic (e.g. tczh). The URL address and the port must match the configurations of the TripleWave instance (use here the ports 4038, 4039, 4040 as defined above):
    - node pathToFolder/WebSocketProducerBig.js –t=tczh –u=ws://localhost:4040 –p=/triplewave/primus
- As the whole cluster is now set up, the TCZH Fast Connector can now be started:

- Node pathToFolder/TCZHFast.js –t0=tczh-original0 –t1=tczh-original1 –t2=tczh-original2

Appendix C: How to connect to the RDF streams

In Section 4.4 we list the five data sets which are fetched, transformed and streamed out to the Web. In this section we describe how these transformed RDF streams can be consumed. We publish the streams over SSE/EventSource and over WebSockets. The components EventSource and WebSocketClient are simple examples of how the streams can be consumed. We show how they can be run.

As we transform five OGD sets there are five RDF output streams. They are available under the following links[57] when using WebSockets respectively the WebSocketClient component:

- CPZH: ws://rdfstreams.ifi.uzh.ch:4041/kafka/primus?topics=cpzh
- ZR: ws://rdfstreams.ifi.uzh.ch:4041/kafka/primus?topics=zr
- TCZH: ws://rdfstreams.ifi.uzh.ch:4041/kafka/primus?topics=tczh
- CPBDE: ws://rdfstreams.ifi.uzh.ch:4041/kafka/primus?topics=cpbde
- CPMDE: ws://rdfstreams.ifi.uzh.ch:4041/kafka/primus?topics=cpmde

If it is desired to connect with the same WebSocket connection to several topics, then add a comma separated list of topics to the URL query, e.g. http://rdfstreams.ifi.uzh.ch:4041/kafka/primus?topics=cpmde,cpbde,tczh

When using the EventSource interface, the streams are available under the following links:

- CPZH: http://rdfstreams.ifi.uzh.ch:6917/cpzh
- ZR: http://rdfstreams.ifi.uzh.ch:6917/zr
- TCZH: http://rdfstreams.ifi.uzh.ch:6917/tczh
- CPBDE: http://rdfstreams.ifi.uzh.ch:6917/cpbde
- CPMDE: http://rdfstreams.ifi.uzh.ch:6917/cpmde

If it is desired to connect with the same SSE connection to several topics, then add a comma separated list of topics to the URL query, e.g. http://rdfstreams.ifi.uzh.ch:6917/cpmde,cpbde,tczh

As an example of a WebSocket client application we implement the WebSocketClient which simply connects to a stream and then prints the data to the console. It can be run using the following command (this command connects to the CPZH RDF stream):

- node pathToFolder/WebSocketClient.js –u=ws://rdfstreams.ifi.uzh.ch:4041 –p=/kafka/primus –q=?topics=cpzh

If one wants to test the connection using the Google Chrome browser there can be found a WebSocket client extension here[58]. After installing it, the desired link above can be used to connect to the stream.

The SSE streams can be accessed by implementing the EventSource interface. We provide a simple component, named EventSource, which connects to the stream and prints the data to the console. Additionally, most browsers have also implemented the interface and so the SSE streams can be consumed by calling one of the corresponding links above in a browser. The EventSource component is run using the following command (this command connects to the CPZH RDF stream):

- node pathToFolder/EventSource.js –u= http://rdfstreams.ifi.uzh.ch:6917 -t=/cpzh

[57] We use ws://rdfstreams.ifi.uzh.ch and http://rdfstreams.ifi.uzh.ch as placeholder for the address

[58] https://chrome.google.com/webstore/detail/simple-websocket-client/pfdhoblngboilpfeibdedpjgfnlcodoo (accessed 1.7.2017)

Appendix D: Content of the CD

- English Abstract (Abstract.txt)
- German Abstract (Zusfsg.txt)
- Bachelor Thesis as PDF (Bachelorarbeit.pdf)
- Folder named Code with four subfolders:
    - Apache-Kafka
    - Components
    - Evaluation
    - TripleWave

The folder named Code contains a README file, which contains more information about the structure and content of this folder.