Publishing Linked Data Event Streams

Andreas De Witte[1] Pieter De Clercq[1] Robin Devos[1] Sieben Veldeman[1]
Harm Delva[2] Dwight Van Lancker[2] Pieter Colpaert[2]

[1]Faculty of Engineering and Architecture, Ghent University
[2]IDLab, Department of Electronics and Information Systems, Ghent University imec

December 20, 2020

Abstract

The Web in its current state cannot easily be interpreted by machines, as the majority of data resources and APIs are documented solely in natural language. This problem can be solved by augmenting resources on the web with semantic metadata; forming the so-called Semantic Web. An open problem is finding a generic approach to process such data in a streaming fashion and propagating updates to data reusers. An idiomatic solution to this problem should build on top of established Semantic Web standards, such as HTTP and RDF. This paper focuses on getting updates to the end-users as soon as possible, for which the applicability of existing technologies has been investigated. Our findings show that this subject can be further divided into four subtasks: (i) formatting, (ii) versioning, (iii) delivery, and (iv) caching. Although every subtask can be solved using existing technologies, none are sufficient to solve the larger problem. We conclude that the solution must lie in the combination of these existing technologies, and that future research should focus on end-to-end solutions – and not only the smaller subtasks.

This paper is available at the following url:
https://thepieterdc.github.io/publishing-lod-streams/.

1 Introduction

On top of the standard Web, which is made for humans, resides the *Semantic Web*. This extension of the Web is used by machines to interpret documents on the web and thus extract information. The Semantic Web is not just about putting data on the Web though, linking to other resources so that more data can be discovered is just as important. Tim Berners-Lee outlined the four principles of *Linked Data* [8]: (i) use URIs as names for things, (ii) use HTTP URIs so that people can look up those names, (iii) when someone looks up a URI, provide useful information using standards such as *Resource Description Framework (RDF)* [28], and (iv) include links to other URIs so that they can discover more things.

This paper focuses on a specific problem that can be solved using Linked Data. This problem involves finding a generic approach for both publishing and updating datasets. Publicly available datasets[1] such as public transport schedules and parking spot availability are inherently real-time. Consequently, receiving an update even 30 seconds too late could already render the data obsolete. However, not every dataset incurs the same volatility. An example is the registry of postal addresses, which may change due to the merging of municipalities, or the currently

[1]Example dataset which counts the number of bikers on several locations in Brussels. https://data.gov.be/en/dataset/d4961387-23f4-44f8-b2e7-c7e9a260a83e

relevant COVID-19 datasets[2], which are updated once per day. Both kinds of datasets have changes that need to become available to possible client applications – as soon as possible, as the change indicates an invalidation of the old dataset.

Deciding the best strategy is influenced by two aspects. The first one is scalability: some technologies are inherently more scalable than others, and these must be compared and evaluated alongside mechanisms such as caching. The second aspect is the volatility of the dataset: data that is changed frequently can benefit from using a streaming-based approach. These aspects have led to an increase in research interest involving Linked Data streaming [35].

In this paper, we present the state of the art regarding this subject. First, the formatting of the data is discussed, together with an overview of some of the RDF serialization formats. The following section considers the versioning of data, namely why this is important and how to execute it. The next section discusses the delivery of data and how publishers can get their data to end-users, together with the criteria to choose an approach. Afterward, caching of data is discussed, different caching strategies for linked data are considered and compared. Finally, the last section applies all of the aforementioned subjects to the use-case of data streaming.

2 Semantic Web

The term *"Semantic Web"* was first introduced by Tim Berners-Lee in 2001[7]. The Semantic Web was established to solve the main problem of the Web, being that the Web is readable for humans, but not for machines. With the Semantic Web, machines should be able to interpret the Web. These machines are called intelligent agents and they will be able to fulfill complex tasks autonomously.

To achieve this goal, a preliminary step is that publishers of data must be able to assign a meaning to the elements of the dataset. More specifically, machines must be able to understand this meaning without human intervention. Since datasets can have a variety of natures and topics, there is a need for so-called *ontologies*, which can be compared to vocabularies for humans. However, vocabularies define terms and ontologies define how these terms are related, mostly for reasoning. Finally, the Semantic Web must be *decentralized*, which means that data should not be contained on a single or a few servers[7].

2.1 Semantic Web Stack

The architecture of the Semantic Web is based on a hierarchy of technologies. This hierarchy is composed as a bottom-up approach, in which every layer uses the layer(s) directly beneath to achieve its goal. This strategy is visualized in Figure 1. Since the subject of this paper is the publishing of data, only the layers that are relevant to this extent are considered. In the figure, this corresponds to the bottom five layers (indicated in bold).

[2]Belgian COVID data. This data is updated regularly, but not in real-time. https://epistat.wiv-isp.be/covid/

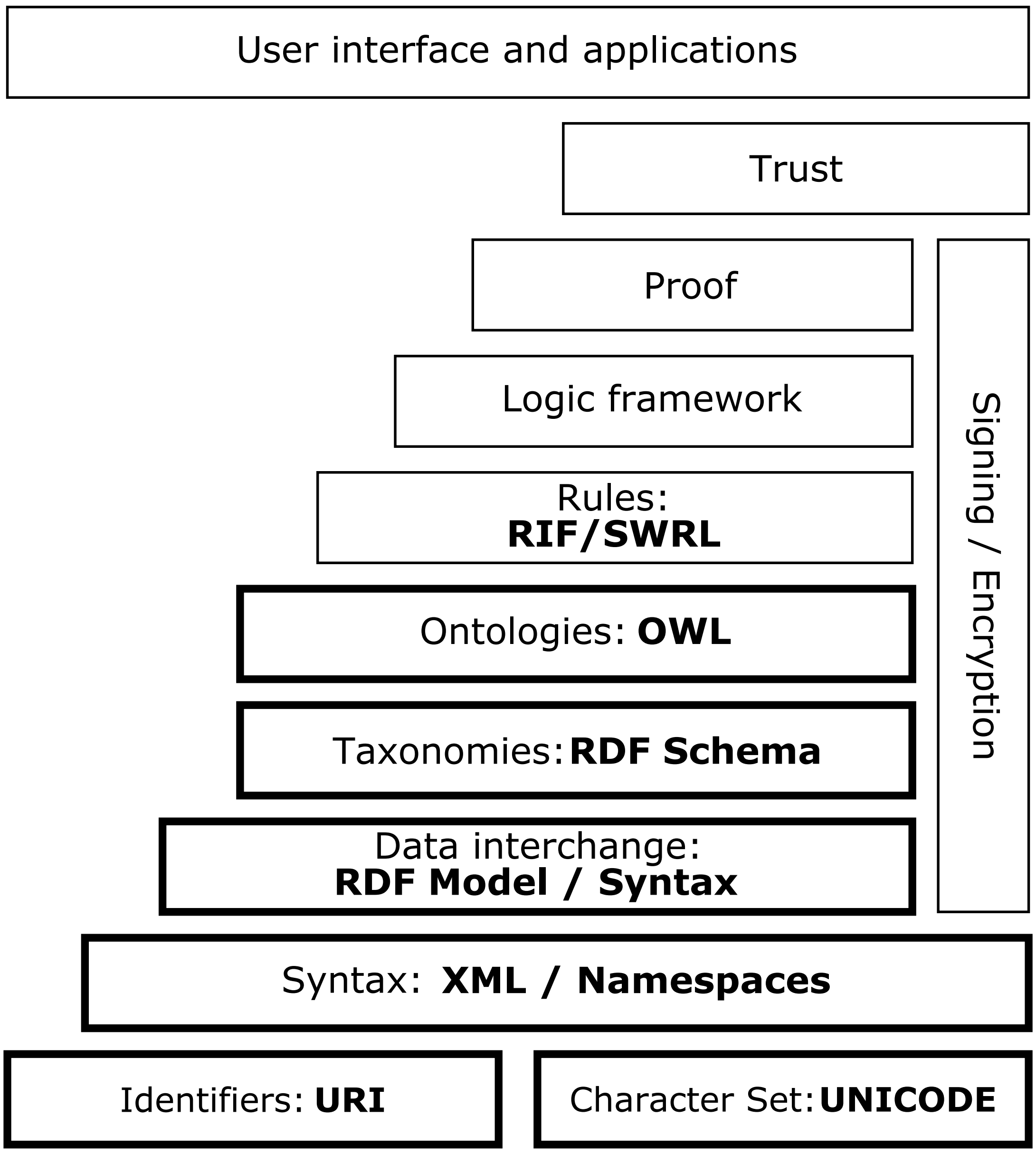

Figure 1: Semantic Web Stack (based on [6, 13]). This image reflects the original Semantic Web Stack from 2001. As a result, some of the technologies are outdated or deprecated (e.g. URI and XML). The relevant technologies for this paper are indicated in bold.

2.1.1 Unicode

Unicode[3] is a system that is used to guarantee a consistent encoding, representation, and handling of text. Similar to ASCII, Unicode was established to aid developers in the creation of applications. However, the advantage of Unicode is that it solves the problems that exist in previous encoding schemes, such as the inability to encode all characters. This problem in particular is tackled by assigning an identifier to each character on every platform, for every program, and in every language.

[3] https://unicode.org/standard/standard.html

2.1.2 Uniform Resource Identifier (URI)

A URI offers a uniform way to identify an object. This identifier is often confused with a *Uniform Resource Locator (URL)*, which *locates* an object, rather than *identifying* it. Consequently, every URL is an example of a URI, but not vice versa[4].
Together with Unicode, which was a W3C recommendation for the Web, URI resides at the foundations of the Semantic Web Stack. This combination enables the identification of resources on the Web uniformly.

2.1.3 Extensible Markup Language (XML)

XML is used to describe data. The most important traits of XML are that it is a flexible, simple and extensible format to store data. XML defines elements using so-called "tags", with every element having both a start-tag and an end-tag. Furthermore, XML supports nested elements, enabling the creation of hierarchies. Since it was a W3C recommendation, XML is at the lower part of the Semantic Web Stack.

2.1.4 Resource Description Framework (RDF) Model, Syntax and Schema

The final element of the Semantic Web Stack is RDF, which is used to add meta-information to a dataset in a describing way. Due to the importance of RDF for the rest of this paper, it is explained in detail in section 3.1.

3 Data formatting

In this section, different ways to format and serialize RDF data are discussed. RDF data can be formatted into multiple formats, such as RDF/XML[30], RDFa[26], Turtle[10], N-Triples[11] and JSON-LD[41]. Each of these formats have their advantages and drawbacks, a detailed comparison is provided further in this section. Additionally, it is possible to receive data from various sources, such as databases. This section elaborates how this data can be transformed into RDF data so that it can be used for Linked Data querying.

3.1 Resource Description Format (RDF)

Before considering the different formats to serialize RDF data, the RDF framework is described. The purpose of RDF is to create a general method to describe resources and relations between data objects, without making any assumptions regarding the domain. The concept of RDF originates from the problem that the World Wide Web was made to be interpreted by humans instead of machines. RDF proves to be an effective way of integrating information of different sources by decoupling the information from its original scheme. In other words, RDF aims to make the Web interpretable by machines[28].

One of the new features introduced in RDF 1.1[5] is the concept of named graphs. Since RDF 1.1, an RDF dataset may have multiple named graphs and at most one unnamed default graph. This enables grouping the data and as such giving extra context about how certain elements of the data are related [36]. Subsequently, named graphs make it possible to both assign a URI to a collection of triples, and to create statements on the whole set.

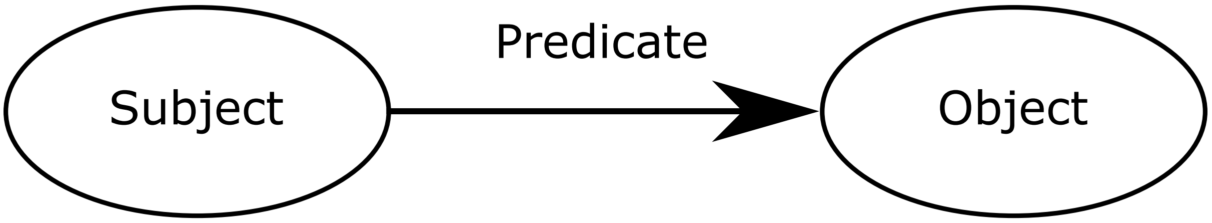

Figure 2: Visualization of an RDF triple as a directed graph. This shows how a triple connects two labeled graph nodes via a directed labeled edge.

3.2 Data model

RDF is structured as a collection of triples. These triples can be visualized as a node-arc-node link, which consists of a subject, a predicate, and an object (Figure 2). Thus this collection of triples can be interpreted as a graph, with the arc pointing towards the object [12].

An RDF triple represents a simple sentence. A commonly used example is the triple (`Alice - Knows - Bob`), which means that the subject "Alice", knows the object of the triple "Bob".

3.3 Serialization format

When reasoning about RDF, it is important to note that RDF itself is not a data format, but a data model. This model describes that the data should be represented as a triple, in the form of `subject - predicate - object`. Hence, to publish an RDF graph, it must be serialized using an RDF syntax[25]. Some of the standardized syntaxes are described below, each with the corresponding notation of the `Alice - Knows - Bob` example.

3.3.1 RDF/XML

The *RDF/XML* syntax is the original RDF serialization format to publish Linked Data on the web. This syntax presents the RDF data model using XML (Example 1). However, the popularity of this syntax is decreasing, because it is difficult for humans to interpret and write [30].

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:about="http://example.org/alice">
    <foaf:knows rdf:resource="http://example.org/bob"/>
  </rdf:Description>
</rdf:RDF>
```

Example 1: RDF/XML representation of the `Alice - Knows - Bob` example

[4] https://www.w3.org/Addressing/URL/uri-spec.html
[5] https://www.w3.org/TR/rdf11-concepts/

3.3.2 Turtle and TriG

Turtle is a serialization format for RDF data that exposes the data in plain text. This format provides prefixes for namespaces and other abbreviations. These prefixes are described at the top of the document. Additionally with Turtle, every triple has to end with either one of three defined suffixes. The first suffix is a comma (`,`). This suffix indicates that the next triple has the same *subject* and *predicate* as the current triple, requiring nothing but the object on the next line. Alternatively, when a triple ends in a semicolon (`;`), the next triple has the same subject (but a different predicate) as the current one. Finally, a triple can have a dot (`.`) as its suffix. This suffix signals that the following triple does not have anything in common with the current triple. The abbreviations of Turtle are non-mandatory. As far as legibility is concerned, Turtle is considered one of the most readable and writable formats [10]. This is illustrated in Example 2.

TriG is an extension of Turtle. Just like Turtle, TriG defines a textual syntax for RDF that allows an RDF dataset to be written in a compact and natural text form. TriG extends the functionalities of Turtle by bringing the possibility to group triples into multiple graphs. Furthermore, these named graphs (as described in Section 3.1) can be preceded by their names. This is so important because named graphs are a key concept of the Semantic Web [39].

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

<http://example.org/alice> foaf:knows <http://example.org/bob> .
```

Example 2: Turtle representation of the `Alice - Knows - Bob` example, from W3.org (`https://www.w3.org/TR/turtle/`).

3.3.3 N-Triples

The N-Triples format (visualized in Example 3) is a subset of Turtle that lacks support for prefixes and abbreviations. Consequently, this format is susceptible to both redundancies and larger file sizes when compared to Turtle. However, this redundancy also has an advantage: it allows line-by-line processing of N-Triples files. This allows for random access, straightforward compression, and usage in streaming contexts. Currently, the format is mostly used for files that are too big to fit into memory [11].

```
<http://example.org/alice> <http://xmlns.com/foaf/0.1/knows> <http
    ://example.org/bob> .
```

Example 3: N-Triples representation of the `Alice - Knows - Bob` example.

3.3.4 JSON-Linked Data (JSON-LD)

JSON-LD is a lightweight Linked Data format, based on the widely used JavaScript Object Notation (JSON) format for formatting data. Because of this and because of its popularity, legibility, and writability, JSON-LD is the ideal format to pass Linked Data in a programming environment. Because JSON-LD uses the same syntax as JSON, it can easily be used to parse and manipulate RDF data [41]. An example of this syntax is provided in Example 4.

```
[
  {
    "@id": "http://example.org/alice",
    "http://xmlns.com/foaf/0.1/knows": [
      {
        "@id": "http://example.org/bob"
      }
    ]
  }
]
```

Example 4: JSON-LD representation of the `Alice - Knows - Bob` example.

3.3.5 RDF in webpages

Multiple formats can be used to embed RDF triples into existing webpages on the internet. The first format is RDFa, which integrates RDF triples into Hypertext Markup Language (HTML) documents by binding the RDF triples with the HTML Document Object Model (DOM). This means that the existing content of pages is shown, thanks to RDFa, inside the HTML code. Because of this, the structured data is exposed to the web [26], as shown in Example 5.

```
<div xmlns="http://www.w3.org/1999/xhtml"
  prefix="
    foaf: http://xmlns.com/foaf/0.1/
    rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
    rdfs: http://www.w3.org/2000/01/rdf-schema#"
  >
  <div typeof="rdfs:Resource" about="http://example.org/alice">
    <div rel="foaf:knows" resource="http://example.org/bob"></div>
  </div>
</div>
```

Example 5: RDFa representation of the `Alice - Knows - Bob` example.

Another option is to use Microdata. This format augments HTML webpages with specific machine-readable labels. It allows nested groups (items) of name-value pairs to be added to documents in addition to the existing content. To create an item, the `itemscope` attribute is used. An extra property can be added to an item with the `itemprop` attribute [32]. This practice is demonstrated in Example 6.

```
<div>
  <div itemtype="http://www.w3.org/2000/01/rdf-schema#Resource"
      itemid="http://example.org/alice" itemscope>
    <link itemprop="http://xmlns.com/foaf/0.1/knows" href="http://
        example.org/bob" />
  </div>
</div>
```

Example 6: Microdata representation of the `Alice - Knows - Bob` example.

JSON-LD (section 3.3.4) snippets can be used to structure data into HTML as well. This method uses a JavaScript notation, embedded in a `script` tag. In other words, within the `script` tag, additional data can be represented in a structured format, using JSON-LD.

3.3.6 Comma-Separated Values (CSV) on the web

CSV is a popular format for publishing data. It is understandable by both humans and machines and it is typically presented in a table because of its structured format. Additionally, CSV data can easily be transformed into RDF data. However, one of the disadvantages of CSV is the absence of a mechanism to indicate the datatype of a specific column, which makes the data hard to validate. CSV on the Web solves this problem by augmenting the dataset with metadata that allows the publisher to provide additional information about the data [45]. Example 7 illustrates how the format of a country code field can be specified.

```
{
    "titles": "country",
    "datatype": {
      "dc:title": "Country Code",
      "dc:description": "Country codes as specified in ISO 3166.",
      "base": "string",
      "format": "[a-z]{2}"
    }
  }
```

Example 7: CSV on the Web example of a country code field specification, from W3.org (https://www.w3.org/TR/tabular-data-primer/).

3.3.7 Protocol Buffers

Protocol Buffers are a technique developed by Google, designed for serializing structured data. Protocol Buffers aim to offer an interface description language for data, which is both simple and offers high performance. The data structures (referred to as messages) and services are described in “proto definition files”, which are programming language-agnostic. Subsequently, Google provides code generators that allow generating source code in multiple different programming languages from these definition files. To achieve high performance, the messages are serialized into a binary format for transmission. This makes them both compact and forward- and backward-compatible. Backward compatibility means that a change in the technology will not break older versions. Forward compatibility on the other hand, means that a file from a more recent version can still be processed by older versions. A disadvantage of Protocol Buffers might be that it was developed for internal use, thus it was not optimized for unpredictable data (such as Linked Data) [27].

In addition to Protocol Buffers, Google has also developed Flat Buffers. These offer access to the serialized data without requiring parsing or unpacking. Furthermore, Flat Buffers focus on memory efficiency, speed, flexibility, a tiny code footprint, strong types and ease-of-use. Flat Buffers work with cross-platform code without dependencies. However, this technology does not seem very useful for Linked Data, because it conflicts with the key principles of Linked Data by enforcing a scheme. Therefore, this is not further discussed in this paper [20].

3.3.8 Header Dictionary Triples (HDT)

HDT is a binary format to represent RDF data in a compact way. An HDT file combines three parts. The first part is the header, which provides metadata about the dataset in a plain RDF format. The second part is the dictionary, which brings a mapping between strings and unique identifiers. The last part contains the triples and encodes the RDF graph using unique identifiers [24].

3.4 RDF*

RDF* (pronounced as "RDF star") is an extension to the Resource Description Framework, which enables RDF graphs to represent interactions and attributes by embedding triples. By nesting triples, an entire triple can become the subject of a second triple. This eliminates the need for intermediary entities, making the model easier to understand. Even though RDF* offers many benefits, it is still under consideration by the W3C and is not yet officially accepted as a standard.

3.5 Decoupling data

Since RDF is used to decouple data from its scheme, it should be possible to receive an arbitrary dataset (structured as e.g. CSV, XML, JSON or in a database), and transform this into an RDF format. To facilitate this, the RDF Mapping Language (RML) is used to materialize data into RDF triples.

RML is defined to express customized mapping rules from heterogeneous data structures and serializations to the RDF data model. Furthermore, RML is a superset of the W3C-standardized mapping language (R2RML). RML provides a generic way of defining the mappings that is easily transferable to cover references to other data structures. Thus, RML is a generic approach, but offers case-specific extensions. This makes it possible to transform any input (e.g., CSV, JSON, relational databases, etc.) into RDF data [16].

4 Linked Data

One of the most critical aspects of the Semantic Web is the ability to create links between multiple datasets. These links are described in a standardized framework, RDF, as previously described in Section 3.1. Not only do these links allow humans to traverse the web, but they also provide a similar pathway for machines to traverse the web as well.

To qualify as Linked Data, the following four conditions need to be met [8]:

1. Objects have to be identified through a unique resource identifier (URI).
2. These URIs have to be HTTP URIs.
3. Data has to be found when we follow such a URI. This data can be formatted in many ways, such as XML.
4. The data has to include links to other data. These links are vital to establishing a "web" of data.

A simple way to represent Linked Data is by using a directed graph, such as visualized in Figure 3. In this figure, the vertices represent objects or data points, whereas the edges represent the links that connect objects or data points. Among other things, this example visualizes the aforementioned `Alice - Knows - Bob` relation. Additionally, the object Alice has a `Name`-link to a data point with the value "Alice". From this, we can infer that the name of the object is Alice.

Finally, observe that every object has at least one link, which implies a connected "Web of Data".

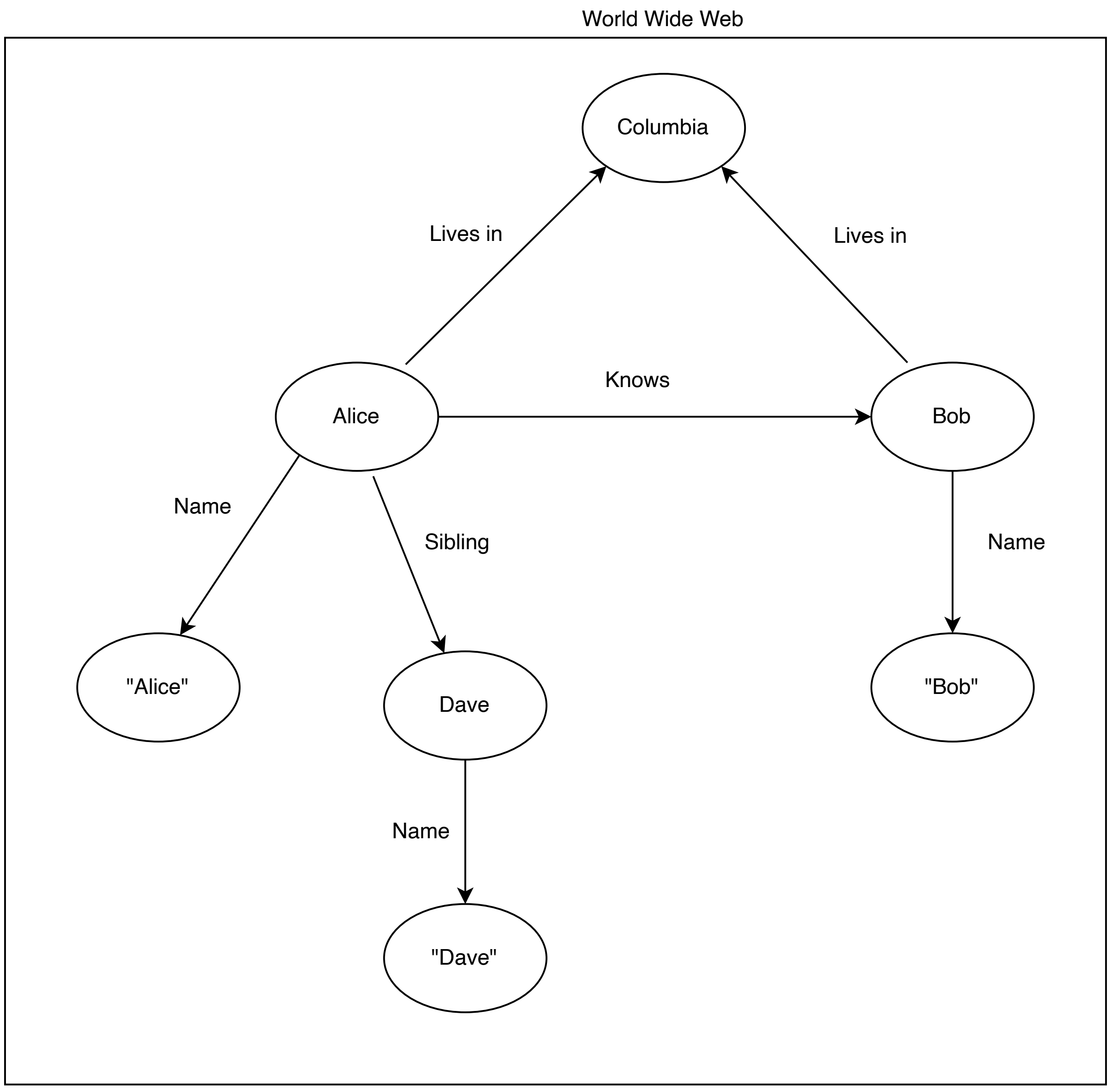

Figure 3: Example of a linked data web.

4.1 SPARQL

SPARQL [40] (a recursive acronym that stands for the SPARQL Protocol and RDF Query Language) is a semantic query language that allows to query and edit data stored in the RDF format. It is the standard query language for RDF and was adopted by W3C[6] as a recommendation in 2008. SPARQL is similar to SQL[7], but where SQL operates on relational databases, SPARQL operates on the Semantic Web. While the functionality is not completely the same, the style of querying for information is very similar, as shown in Example 8.

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
    ?person foaf:name ?name .
}
```

Example 8: SPARQL example that lists the names of all the objects that have this link, from W3.org (https://www.w3.org/2009/Talks/0615-qbe/).

5 Data versioning

Data versioning has two main advantages for both the data consumers and the providers. The first advantage for consumers is the ability to consider only data in a specified timeframe of interest. For example, a train schedule application does not require information about interruptions and delays from a month ago. Versioning allows the provider to define so-called "snapshot" versions, which correspond to checkpoints of the dataset. Consumers can reconstruct subsequent versions of the dataset by fetching only the introduced changes (*deltas*), since a given snapshot. The benefit of publishing these deltas is that the consumers do not need to download the entire dataset every time an update is published, they can download only the changes. For the providers, storing these deltas eliminates the need to store duplicate or redundant information.

5.1 Git-based

The first type of approach that is discussed is derived from the world of software engineering. Software developers use version control systems, such as Git or Mercurial, to work on the same source code files collaboratively. The same idea can be used to version data, as proposed by Arndt et al.[3]. This paper formalizes the requirements for a git-based data versioning infrastructure as follows:

1. **Support divergence of datasets:** The more users that contribute to the dataset, the higher the probability that some users disagree on how the dataset should evolve. Consider for example a dataset of train connections in Belgium. The set of contributors might be divided into smaller working groups, each with their dedicated focus. Some users might prefer to work on train stations in Ghent, others in Antwerp, etc. This concept corresponds to *branches* in Git.

2. **Conflate diverged datasets:** Once the working groups have finished their subtasks, they should be able to integrate their changes back into the upstream dataset. This is achieved through a *merging* process, similar to Git. When merging two branches, conflicts may arise in case other branches that touch the same data have already been integrated. The

[6] https://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation/
[7] https://www.w3schools.com/sql/sql_quickref.asp

system must support both conflict detection and resolution to prevent compromising data integrity.

3. **Synchronization of derivatives:** To enable distribution of the dataset across multiple servers, a synchronization mechanism is required to ensure every device has a consistent version of the dataset. In Git, this corresponds to having both a local copy of the repository and a remote repository (via e.g. GitHub[8]). Subsequently, the local dataset can be updated via *pull* operations, whereas local changes can be *pushed* to the remote repository.

5.2 RDF archives

Another approach that is gaining increased interest among researchers, is RDF archives [19, 18]. This approach describes a technique for archiving RDF data and executing queries on those archives. RDF archives entail six different query types and three different storage strategies.

5.2.1 Query atoms

The corresponding literature currently distinguishes six different types of queries, called *Query atoms*. These are explained below using the example of a car park.

1. **[VM] Version materialization:** The first type of query is the most basic form of retrieval. Version materialization allows the consumer to obtain the complete state of the dataset at a given timestamp or version, similar to the earlier discussed *snapshots*. Note that this technique is also being used by existing web archiving services, such as the WayBack Machine of the Internet Archive[9]. In the context of the car park example, this is equivalent to listing all cars that are in the car park at the given timestamp.

2. **[DM] Delta materialization:** Subsequently, the consumer might be interested in obtaining the differences between two versions. This request is fulfilled using the second type of query. In order to enable this functionality, the system must support conflating several deltas into a new version to calculate the net differences. In the car park example, this would allow the consumer to retrieve a list of all cars that have entered the car park today.

3. **[SVQ] Single-version structured queries:** Next, literature considers the evaluation of complex queries on one specific version of the dataset. This functionality is currently offered by SPARQL (Section 4.1) endpoints. An example query on the example situation could be: "is there a blue car in the car park on January 1, 2020?".

4. **[SDQ] Single-delta structured queries:** The same analogy as with the first and second type of query can also be applied here. As an example, consider the following query: "has a blue car entered the car park today?".

5. **[CVQ] Cross-version structured queries:** The fifth type is very similar to the third type, except that the goal is to query across multiple versions at once. The aforementioned example query can be slightly adapted as follows: "has there ever been a blue car in the car park?".

6. **[CDQ] Cross-delta structured queries:** Finally, the previous query type also has a delta counterpart. The above query can also be modified to serve as an example for this type: "on average, how many cars enter the car park per day?".

[8]https://github.com/
[9]https://web.archive.org/

5.2.2 Storage

Depending on which query atoms should be optimized, a different storage strategy is preferred.

1. **[IC] Independent Copies:** The first technique simply creates a separate, isolated instance of the entire dataset every time a change is introduced. The advantage of this storage technique is high performance for 'VM', 'SVQ' and 'CVQ' queries, but the downside is twofold. First of all, the duplication of the datasets inevitably incurs scalability issues. Additionally, since deltas do not exist, these must be calculated on-the-fly in order to evaluate delta-oriented queries. This task might be computationally expensive.

2. **Change-based approach:** The second technique relates more to the aforementioned Git-based approaches. This technique starts from an empty dataset and stores the changes (deltas). Consequently, this strategy addresses the scalability issues of the previous approach. Furthermore, this technique is very efficient to evaluate delta-based queries (`DM`, `SDQ`, `CDQ`), but requires these deltas to be conflated every time a version-based query is evaluated.

3. **[TB] Timestamp-based approach:** The final strategy augments the stored triples with temporal validity information and acts therefore as a combination of the two other strategies. This approach supports compression to reduce space and can be used for all queries except `DM`.

5.2.3 OSTRICH

In addition to the above three storage techniques, Taelman et al. [43, 44] proposes *OSTRICH* as a versioned "hybrid IC-CB-TB" storage mechanism. Because of this, OSTRICH is able to efficiently evaluate `VM`, `DM`, `SVQ` and `CVQ` queries and return the results as a stream of triples. Being a hybrid storage mechanism, the working of OSTRICH is inspired by combining the best properties and ideas of the previously discussed techniques. First, an immutable copy of the dataset is made, which serves as the initial version. This version is saved as an `HDT`-file (section 3.3.8). This format is a binary RDF representation that features both a high compression ratio, as well as indices to "enable the efficient execution of triple pattern queries and count estimation". All subsequent updates to the dataset are stored as deltas, which are also indexed and merged according to their timestamp to consume less disk space.

The advantage of using OSTRICH over the aforementioned Git-based approach is that OSTRICH makes it possible to execute version-based queries, without requiring a materialization step. This vastly speeds up the entire query.

6 Data delivery

In this section, the transmission of the data to the users is discussed. After the previous sections have described how data can be stored, the next problem that arises is how to expose it for consumers to access. The most commonly used protocol for web data exchange is HTTP, which uses a `request-response` communication pattern, in which an HTTP server answers to the requests of a client sequentially. However, this might not be the most optimal form of communication in the context of this paper, since this requires the consumer to send a request each time an update is desired. Consider on the other hand the `publish-subscribe` pattern, in which updates are pushed to the consumer (*subscriber*) automatically when they are available, without requiring an explicit intent from the consumer. This pattern is implemented in the MQTT protocol and in existing software such as Apache Kafka, which employs a similar strategy and uses a custom protocol directly over TCP.

However, the MQTT protocol is not the silver bullet for data delivery, since it is not built upon HTTP and it is not standardized by the W3C Web of Things either. Instead, it was designed for IoT (*Internet of Things*) applications, which concerns primarily machine-to-machine communication, not on the Web. The protocol itself is therefore not useful in this context, but the characteristics can be used.

The existing HTTP protocol can still be a viable option since it can serve as the foundation on top of which a new protocol can be built. Alternatively, an already existing protocol with suitable characteristics can be adapted to better meet the needs of event-based processing.

6.1 Delivery Approaches

When devising a strategy to deliver a given Linked Open Dataset, some considerations must be taken into account. The first aspect is the partitioning of the data, which impacts how it can be transmitted. Secondly, since the goal is to work with event streams, the architecture should be decentralized. Finally, the strategy should offer storage and performance benefits that are comparable to using a centralized, data-dump style approach [15].

In the current state of the art, the nature of the data affects the transmission. In the case of real-time sensor data, for example, Atmoko et al. [4] describe an MQTT-based approach and indicate that this is more efficient than using regular HTTP. However, since the goal is to achieve a generic approach for each kind of data, elements of different technologies should be combined, so that both machines can consume the received data, but that it can also be consulted by humans in a browser.

6.2 Low-level protocols

Various low-level protocols exist, such as Message Queuing Telemetry Transport (MQTT), the Advanced Message Queuing Protocol (AMQP), or the Constrained Application Protocol (CoAP). These protocols are focused on the transmission of data on constrained, low-level devices. Wang [47] shows a similar conclusion when comparing MQTT and HTTP on data from IoT devices. Some datasets used by Van de Vyvere et al. [46] also consist of IoT data. While, in this context, the intention is not to use a lightweight, lower level protocol such as MQTT (instead the goal is to publish data on the web, using HTTP), it is valuable to look at what makes it the best option, i.e. the different communication model, and adapt the desired technology accordingly.

Furthermore, low-level protocols such as MQTT suffer from security vulnerabilities[10], which is problematic in the context of open data publishing. HTTP on the other hand, is well established, mainstream, and therefore less susceptible to the same kind of problems. Hence from a security point of view, it makes sense to prefer HTTP over low-level protocols.

6.3 HTTP-based approaches

Other options include the use of *web feeds*, such as RDF Site Summary (RSS). RSS publishes updates in a feed and allows users to access them in a standardized format[11]. However, RSS is

[10] https://www.cvedetails.com/google-search-results.php?q=MQTT
[11] https://www.xml.com/pub/a/2002/12/18/dive-into-xml.html

an umbrella term that spans different formats. Therefore, Atom [22, 38] was created, to achieve more standardization and disambiguation. Atom uses a separate protocol on top of HTTP. These two similar approaches to data publishing can both be considered since the concept and use of web feeds are similar to the goal of the event-based approach.

Linked Data Notifications [23, 9] (LDN) is another protocol developed by the W3C. This protocol shares similarities with the aforementioned Kafka and MQTT technologies. Where Apache Kafka uses a *broker* as an intermediary service between Senders and Consumers, LDN uses an *inbox*.

A *sender* can send a notification to a *receiver* using an HTTP `POST`-request. The receiver is then responsible for storing these in their inbox. When a consumer wants to fetch these notifications, they can send an HTTP `GET`-request to the URL of this inbox. The response to this request contains a listing of all the corresponding notifications. Each entry in the list must be formatted as an RDF source and contains a URI, which the consumer can subsequently use to query specific notifications from the inbox. According to the specification, each notification needs to at least be formatted with the JSON-LD (section 3.3.4) `Content-Type`, but additional serializations are allowed as well. This clearly shows the similarities with a system that uses brokers. However, LDN seems less focused on continuously updating data and speed and more on adaptability and the ability to use it in a variety of contexts [9]. Evaluating the performance of LDN, when speed is crucial, can be a topic for further research.

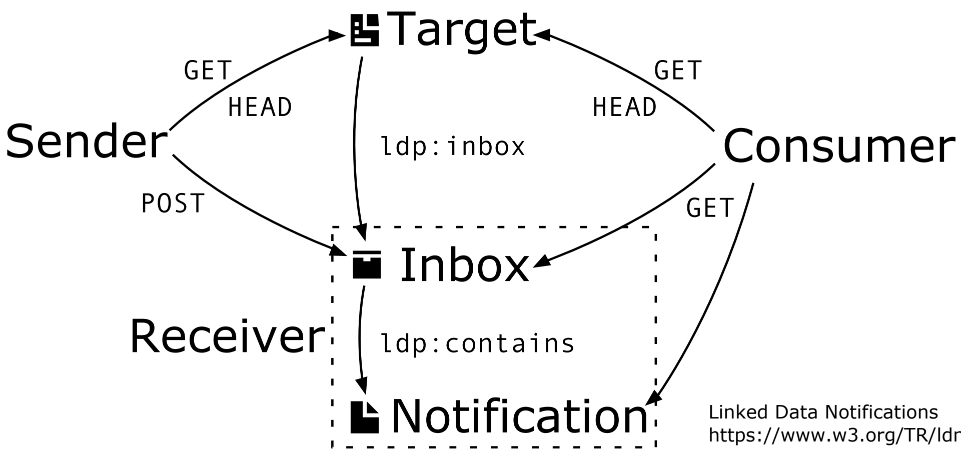

Figure 4: Linked Data Notifications delivery model [9]. Note the different HTTP requests indicated using the arrows, as discussed in the text. Also note the discovery of the inbox and notifications using RDF predicates.

Van de Vyvere et al. [46] compare an HTTP polling approach with a push-based approach using *Server-Sent Events (SSE)* on live Open Datasets. The datasets that are used are real-time in nature and most of them are almost continuously updated. The authors clearly show that, in order to minimize latency on the client, a push-based approach is required. When using these approaches, the server keeps the connection to the client alive, so that multiple updates of the dataset can be sent over the same connection. This practice eliminates the need for costly handshakes when initiating new connections and consequently benefits the performance. In addition to SSE, the authors also consider *WebSockets*, which uses an HTTP handshake to initiate the channel, but further transmission occurs over a raw TCP connection that supports full-duplex bidirectional communication. SSE and WebSockets offer similar performance and since SSE is unidirectional and only reliant on HTTP, the former was chosen in this article. Moreover, the CPU usage of SSE in comparison to polling was also shown to decrease. In order to find a generic approach for all kinds of Linked Open Datasets, it is clear that some sort of pushing from the server should be supported to ensure support for fast-changing, continuously updating datasets.

WebSub [34] is the last protocol considered in this section. This protocol was adopted by the W3C in 2018 and provides publish-subscribe communication using HTTP. Previously, this protocol was named *PubSubHubbub*, because it uses the concept of *hubs* as intermediary servers between publishers. These publishers own *topics* and push updates to them, whereas they can be subscribed to by *subscribers*. Subscribers need to be network-accessible at all times using their *Subscriber Callback URL*. In order to subscribe to a topic, the subscriber sends an HTTP `POST`-request to a hub that contains their callback URL. For publishers, however, the specification does not provide a standardized way to update the hubs, hence it can be chosen by the developers. When a hub receives an update from a publisher, it notifies the subscribers by relaying the updates to their specified callback URL. The specification recommends that HTTPS is used at all times for HTTP requests and also specifies an additional `X-Hub-Signature` header that can be used for message authentication. This protocol might be of interest since it inherently supports decentralization, namely, hubs can be partitioned according to the needs of the user and so can the topic (i.e. topics can be duplicated across different hubs or could only be present on certain hubs). The downside to this protocol is that the subscribers must always be reachable via HTTP, which implies that the subscriber must operate a web server, to which updates can be pushed.

7 Data caching

This section describes and compares various techniques in literature that can be used to enable caching of large open datasets. Multiple aspects of caching are considered, the first is an overview of pull- and push-based strategies to update caches, and to notify consumers of changes. Next, techniques to cache query results are discussed. This section is concluded by providing a case study of caching on the Web, more specifically in Content Delivery Networks.

7.1 Pull-based approaches

Many different algorithms exist to update large Open Data caches using pull-based approaches, with varying computational complexity. Dividino et al. [17] consider the following five elementary strategies:

1. **Age:** First update the oldest part of the dataset.
2. **Size_SmallestFirst:** First update the smallest part of the dataset.
3. **Size_BiggestFirst:** The opposite of the previous strategy, first update the biggest part of the dataset.
4. **PageRank:** Update the highest-ranked page in the dataset.
5. **ChangeRatio:** Update the part of the dataset that was changed the most (using set difference).

In addition to these simple algorithms, Dividino et al. [17] propose two more complex techniques. These techniques each have two variants, one using the Jaccard distance (ending in `-J`) and the other one with the `-D` suffix which uses the Dice coefficient, taking into account the last two retrieved versions. The first algorithm is *ChangeRate*. This algorithm is similar to the simple *ChangeRatio* algorithm, but uses the Jaccard distance or Dice coefficient instead of set difference. Finally, the authors propose the *Dynamics-J* and *Dynamics-D* algorithms, that update the most dynamic data source. The authors conclude that these final four algorithms work better when there is limited bandwidth.

Nishioka et al. [33] describe how the lifespan of RDF triples can be predicted and how these predictions can be used to update local caches. These predictions are obtained by training a linear regression model on different features of the RDF triple. The first feature is the *Pay-level domain (PLD)* of the subject. For example, the PLD of the subject *https://dbpedia.org/resource/Google* would be `https://dbpedia.org`. As a motivation for this choice, the authors state that triples from the same PLD will have a similar lifespan. Additionally, the predicate and the Pay-level domain of the object are used. If the object is a literal, rather than a URI, the literal value is used instead. When compared to taking the mean average of the frequencies in the training data to all triples in the test set, the predictions provided substantially better results. Using this predicted lifespan, the authors propose an algorithm to update a local cache based on the reciprocal of the mean average of a triple. This algorithm was compared with the best performing algorithm proposed by Dividino et al. [17], which is the *Dynamics-J* algorithm. This comparison was done on the DyLDO and Wikidata datasets with a varying amount of bandwidth. For every bandwidth, both algorithms offer very similar precision, but the precision of the linear regression model is always higher than the *Dynamics-J* algorithm. This difference is the most significant when the bandwidth is low, in which the precision for the linear model is 0.998, whereas the Dynamics-J algorithm has a precision of 0.996. As the relative bandwidth increases, both algorithms achieve a precision of 100%. The authors conclude that their approach outperformed the methods proposed by Dividino et al. [17], and that it requires fewer resources because the previous snapshots of the data are not required. Finally, the authors discussed more advanced machine learning models, which did not incur a significant improvement on the results.

The final pull-based algorithm is *Application-Aware Change Prioritization (AACP)*, using the *Preference Aware Source Update (PASU)* strategy. This algorithm is proposed by Akhtar et al. [2] and combines a change metric and a weight function to identify the recent changes. The authors have compared their algorithm to the already existing solutions described in this section. This comparison has been done on the DyLDO and BTC datasets. When compared to the five elementary algorithms and the *ChangeRate* algorithm by Dividino et al. [17], the authors found that their algorithm outperforms every algorithm with an effectivity of 93.5%, which is 8.8% higher than the second-ranked *ChangeRate* algorithm. Their algorithm was also compared to the aforementioned linear regression model by Nishioka et al. [33], in which their algorithm has higher precision and a higher recall score than the linear model, for every iteration. This difference is the most significant during the first iteration, namely 0.03 for both precision and recall on the BTC dataset.

7.2 Push-based approaches

The approaches described in the previous section rely on pull-based mechanisms. Using these mechanisms, the client is responsible for updating their local cache themselves. Push-based approaches work differently, in the sense that the server notifies clients of changes in the dataset. The main advantage of this strategy is that the amount of unnecessary requests is reduced since clients do not need to poll for updates to the dataset. However, pushing updates to the clients will temporarily result in a bigger load on the server.

Rojas Meléndez et al. [37] discuss the advantages of a *publish-subscribe* approach and use parking data in Flanders as a proof of concept. While the authors did not have a large enough dataset to verify their arguments, they still argue the ease of implementation of a push-based strategy.

Finally, Van de Vyvere et al. [46] compare three metrics for an SSE approach and HTTP polling approaches (with and without nginx[12]). These metrics are latency (the amount of time it takes for an update to be registered by the consumer), memory usage, and CPU usage. The authors conclude that pull-based approaches always incur a higher latency than push-based approaches. On the other hand, more memory is required to use push-based approaches, because all connections to clients need to be kept alive in memory. The authors assumed that CPU usage would be better for a push-based approach initially, but that for a higher amount of consumers, the pull-based approaches would perform better. However, this hypothesis was rejected based on the results. Overall, their comparison shows that a push-based approach is generally the best choice. It is important to remark that the authors have used a simplistic pulling approach for this comparison, as opposed to the more complex approaches described in the previous section.

7.3 Caching query results

The aforementioned approaches always assume that the entire dataset is stored at every consumer. However, if the dataset becomes too large, this is not a viable option. Alternatively, the results of queries can be cached instead. This section describes three approaches to achieve this.

The first cache technique is proposed by Akhtar et al. [1]. This technique uses an adaptive cache replacement strategy that examines query patterns from clients, and prefetches the results of possible future queries accordingly. This study shows an average reduction of query times by 6.34% when compared to existing approaches such as *LRU* (Least Recently Used), *LFU* (Least Frequently Used) and *SQC* (SPARQL Query Caching).

Secondly, Yi et al. [48] propose a two-layer caching mechanism. The first layer implements the *Adaptive Replacement Cache-memory algorithm (ARC)* and operates in memory. The purpose of this layer is to cache all the frequently used queries that are either not complicated, or for which the result set is small. The other cases are handled by the second layer. This layer is a key-value database that is only partially running in memory. The approach has been compared to existing techniques on the OLTP-trace test set. The results prove that the proposed method has a higher hit ratio, more specifically for a cache size of 1000 entries, an LRU approach has a 0.43134 hit rate, while the ARC algorithm achieves a 0.44938 hit rate.

The final method for caching Linked Open datasets is the *CyCLaDEs* approach, proposed by Folz et al. [21]. This approach consists of a decentralized cache that uses a "neighborhood" of clients. Every client in the same neighborhood hosts related fragments in their cache. Subsequently, a new client will first contact clients in this neighborhood before querying the server. This idea assumes that clients that have had similar queries in the past will also perform similar queries in the future. The algorithm first constructs a random subset of clients and then selects the k clients that are the most similar, to compose a neighborhood. Each client uses an LRU cache with a fixed size. It might be useful to combine this decentralized cache system with the two layer cache mechanism described above. The authors conclude that their decentralized approach is able to handle a lot of queries that a single local cache would miss. Furthermore, the CyCLaDEs approach with ten neighbors was compared to existing caching

[12] A high-performant web server and reverse proxy (`https://nginx.org`)

approaches on the BSBM-1M dataset. In both cases, the local caches can handle 40% of the total requests, but in the CyCLaDEs approach, 22% of the requests could be served from the neighborhood. They propose that their method could indeed be used in applications where the load on the main server can be very high. The authors did not explicitly investigate this, but this can be compared in future work.

7.4 HTTP Caching

Caching is used a lot on the Web, mostly in the form of HTTP caching. Given the many different HTTP methods that exist, it only makes sense to cache *safe* requests that do not modify the state of the server, since these cannot invalidate the data. The most prominent examples of these requests are `HEAD`, `GET`, and `OPTIONS` requests, of which `GET` requests are the most common. In order to indicate how a request can be cached, the server can include a `Cache-Control` header in the response. This response has several possible values, multiple can be combined.

1. **no-store:** This response may not be cached by the client.
2. **no-cache:** This response may be cached by the client, but every time it is requested, the client must send a `HEAD` request to the server to validate whether the cached result is still up to date.
3. **private:** This response may only be cached in a web browser and not on intermediate servers. This option can for example be used when the data is confidential.
4. **public:** Every server (including intermediaries) may cache this response.
5. **max-age=x:** The response may be cached for at most x amount of seconds, without revalidating.
6. **must-revalidate:** The response must be revalidated every time it is requested.

In addition to the `Cache-Control` header, the server may include an *etag* in the response that is calculated as a hash of the content. When sending this request again in the future, the client can include an `If-Modified` header with this etag as its value. If the server detects that the etag is still valid for the current version of the dataset, it sends an `HTTP 304 Not Modified` status code, indicating that the cached copy of the client is up to date.

If no caching headers are present in the response, the client may use a heuristic that evicts the response from its cache when 10% of the difference in time between the current timestamp and the timestamp in the `Last-Modified` header is elapsed. This can be compared to previously discussed dynamic based algorithms. If a cached document is expired, the client can either send the `GET` request again and cache the new response, or send a lightweight `HEAD` request to validate the state.

Note that HTTP caching only uses age or dynamics, which corresponds to the pull-based approaches described in section 7.1. It was argued that push-based approaches had a better performance, which is why we propose that further research is required towards applying push-based approaches in HTTP communication.

7.5 Caching in Content Delivery Networks

An example of an intermediary in HTTP caching is a *Content Delivery Networks (CDN)*, which distributes cached copies of webpages over multiple servers that are located around the world. As a result, the perceived performance of a website no longer depends on the distance of the client to the server. Since these networks rely heavily on state of the art caching strategies, it is worth investigating scientific literature that considers Content Delivery Networks and try to apply the findings to Linked Open Data.

The first technique is proposed by Berger [5] and concerns a supervised learning technique to train a lightweight boosted decision tree, using calculated *optimal decisions (OPT)* from the past. This model is subsequently used to update the cache in the present, a technique called *Learning from OPT (LFO)*. The authors state that this model achieves an accuracy of 93% in predicting the OPT decisions, but there is a 20% gap in caching performance between OPT and LFO. So even though the model predicts the best decision in 93% of the cases, its caching performance is significantly lower than the most optimal model. We argue that this method could also be applied as a caching strategy for query results caches of linked open data, but it still needs to be compared to other strategies described in the previous sections.

Current cache replacement algorithms mostly consider the frequency and locality of data. Li et al. [29] propose to use the access interval change rate as an alternative metric to update caches. In this work, a naive algorithm is implemented that uses this access interval change rate and compares it to basic algorithms such as LRU and LFU. The authors show that using the access interval change rate results in a better caching performance. While this method is fairly simple, it is very performant. Combined with the conclusions from section 7.1, we, therefore, conclude that these algorithms are beneficial if a simple strategy is needed, but that even better performance can be achieved with more complex algorithms.

8 Data streaming

In this final section, different aspects are highlighted on how to maintain a stream of linked data. The first section presents a vision that compares the distributed storage of linked data with the current way of storing DNS records, through the use of intermediaries. The subsequent sections describe existing techniques specifically tailored towards linked data streaming, in the context of the earlier discussed formatting, versioning, delivery, and caching methods.

8.1 Use of intermediaries

Remark that we can observe an interesting correlation between chaining SPARQL endpoints and the distribution of datasets, versus the mechanics of *Domain Name Servers (DNS)*. These servers start from a *root* DNS server and delegate the request of the user through multiple intermediaries. The same idea can be applied when scaling linked data streams, or even linked datasets themselves. A root server could for example serve as an entry point for data publishers to push their data to, as well as for users to send their requests to. These requests can subsequently be routed via multiple intermediaries, depending on which part of the data is required. Consider, as an example, that we are interested in a stream of train delays for the Ghent-St. Pieters train station in Belgium. The root server could contain links to several intermediary servers that contain this data for each continent. The next level could divide this further into countries per continent, followed by a city-based level and eventually a station-based level. This way the distribution of the stream can even be scaled adaptively, depending on the need. This idea can be implemented using the topic-based strategy used in section 6.3.

8.2 RDF Stream Processing

The Internet of Things has caused a shift in the data landscape. Sensors publish data at high frequency into the cloud, so that it does not always make sense to use polling-based approaches. Instead, it is much more efficient to treat these values in a push-based (section 6) streaming fashion [14]. To manage these data streams, Stream Processing Engines have seen the light. However, since the contents of the stream can be in all shapes and sizes, a Web of Data-counterpart was needed. This has lead to the creation of RDF Stream Processing techniques, which allow to process RDF-based data streams using various operations such as filtering and aggregating.

8.2.1 TripleWave

TripleWave is an open-source framework that allows the creation and the publishing of RDF streams. In their requirements section, Mauri et al. [31] list seven requirements to which the framework must comply. The most interesting requirements are `R1` and `R2`, which respectively state that "TripleWave may use streams available on the Web as input" and that "TripleWave shall be able to process existing time-aware (RDF) datasets" (which could either be formatted as a stream or not). These requirements imply that the framework has conversion mechanisms to transform this data. The framework uses CSV and JSON connectors to convert existing web streams, as well as R2RML (section 3.5) for the generation of RDF streams.

For consumption of the streams, the framework provides both a push-based and pull approach, using WebSockets, end users can subscribe to a TripleWave endpoint and get updates pushed to them. Alternatively, streams can also be published for the user to pull them.

8.2.2 Web Stream Processors

Dell'Aglio et al. [14] propose an infrastructure that is capable of handling TripleWave (section 8.2.1) as input, and impose seven requirements for a Web of Data Streams, called *Web Stream Processors (WeSP)*. These requirements are inspired by previous work from Stonebraker et al. [42].

1. **Keep the data moving:** The first requirement is that *WeSP must prioritize active paradigms for data stream exchange, where the data supplier can push the stream content to the actors interested in it*. This means that instead of using polling-based approaches, data publishers must push updates to subscribers. This has been discussed in Data delivery (section 6) and matches the findings for low latency updates by Van de Vyvere et al. [46, 37], since streams with high frequency correspond to real-time data flows.

2. **Stored and streamed data:** As a second requirement, *WeSP must enable the combination of streaming and stored data*. Stored data in this context refers to data which has been formatted using the open data standards described in Data formatting (section 3), such as RDF. Additionally, this data can be exposed and queried via existing SPARQL-endpoints. To allow both streaming and storing the data, one of the RDF archive storage-techniques (section 5.2.2) can be used. Specifically for streaming-based data, the delta-related query atoms are the predominant use case and therefore a change-based approach should be preferred.

3. **High availability, distribution and scalability:** Next, Dell'Aglio et al. [14] require that *WeSP must enable the possibility to build reliable, distributed and scalable streaming applications*. Obtaining high availability is possible using either one of the provided Data versioning (section 5) approaches. The Git-based approaches in particular allow a dataset

to be distributed efficiently across multiple devices and therefore improve the scalability and reliability. Depending on the frequency of the updates, the reliability can be further improved through caching.

4. **Operations on the stream content:** A subsequent requirement is that *WeSP must guarantee a wide range of operations over the streams*. As mentioned before, RDF Stream Processing requires that the stream can be filtered or aggregated and that the result is another stream, which can be observed. This can be implemented via SPARQL queries that receive streamed data as their input, apply the desired operation, and return a new output stream of the results. Multiple of these queries can be chained together, to obtain the desired output depending on the use case.

5. **Accessible information about the stream.** In order to improve the accessibility of the data, *WeSP must support the publication of stream descriptions*. These stream descriptions can contain metadata, such as the expected update frequency and the expected size of the updates. This information can serve multiple purposes. If a Git-based approach would be used to store the data, the meta-information can indicate the parent version, which can subsequently be interpreted to update a local copy by resolving the deltas. Additionally, if applicable, `Expiry` timestamps can be added to benefit from Data caching (section 7).

6. **Stream variety support:** The penultimate requirement is that *WeSP should support the exchange of a wide variety of streams*. Since the web is of decentralized nature, we cannot define a single model and format to which all streams must adhere. Instead, every stream can have its own format, depending on the content type and publishing frequency. Various formats discussed in section 3 can be used.

7. **Reuse of technologies and standards:** Finally, *WeSP should exploit as much as possible existing protocols and standards*. A manifold of existing technologies and infrastructures already exist. In the context of formatting, versioning, delivery, and caching, this paper has provided sufficient examples. These existing technologies should be reused whenever possible to encourage interoperability. While starting from scratch can often give good results on a local scale (e.g. devising a format that is specifically created for the particular use case of the data), but prevents the technology from being globally adopted.

9 Conclusion

The goal of this paper was to investigate how existing technologies can be applied in the field of Linked Open Data to handle data streaming. We have performed a literature study and explained various techniques to define, version, distribute, and cache this data. Multiple existing and viable technologies have been described, and each section stressed that existing technologies could be combined to achieve better performance. A lot of research has been conducted at the protocol level, but further research at higher levels is recommended and may prove beneficial. Furthermore, it was noticed that existing pull-based algorithms, which perform better, have not yet been compared to a push-based approach. Finally, we propose that the technologies described in the different sections could be combined into a solution for distributing live open Linked Datasets with event streaming.

However, every dataset is different and therefore requires a different approach. For example, a dataset with parking spot availabilities needs an approach that prioritizes the speed of distribution of data changes, while an address registry dataset needs to prioritize access to this dataset at different times. We concluded in each section that various techniques exist to solve the subtask discussed in that section. However, no silver bullet exists (yet) that solves the complete problem. Therefore, we conclude that an exhaustive solution must use a combination of existing techniques and that future research should not consider the subtasks in isolation, but instead focus on the end-to-end combination of these subtasks. The benefit of reusing existing technologies, rather than establishing new standards, is the interoperability with existing systems.

References

[1] Usman Akhtar et al. "A cache-based method to improve query performance of linked Open Data cloud". In: *COMPUTING* 102.7 (JUL 2020), 1743–1763. ISSN: 0010-485X. DOI: {10.1007/s00607-020-00814-9}.

[2] Usman Akhtar et al. "Change-Aware Scheduling for Effectively Updating Linked Open Data Caches". In: *IEEE Access* PP (Sept. 2018), pp. 1–1. DOI: 10.1109/ACCESS.2018.2871511.

[3] Natanael Arndt et al. "Decentralized Collaborative Knowledge Management Using Git". In: *Journal of Web Semantics* 54 (Sept. 2018). DOI: 10.1016/j.websem.2018.08.002.

[4] Rachmad Atmoko, R Riantini, and M Hasin. "IoT real time data acquisition using MQTT protocol". In: *Journal of Physics: Conference Series* 853 (May 2017), p. 012003. DOI: 10.1088/1742-6596/853/1/012003.

[5] Daniel S. Berger. "Towards Lightweight and Robust Machine Learning for CDN Caching". In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. HotNets '18. Redmond, WA, USA: Association for Computing Machinery, 2018, pp. 134–140. ISBN: 9781450361200. DOI: 10.1145/3286062.3286082. URL: https://doi.org/10.1145/3286062.3286082.

[6] Tim Berners-Lee. *WWW Past, present and future*. 2005. URL: https://www.w3.org/2003/Talks/0922-rsoc-tbl/.

[7] Tim Berners-Lee, James Hendler, and Ora Lassila. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 34–43.

[8] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked data: The story so far". In: *Semantic services, interoperability and web applications: emerging concepts*. IGI Global, 2011, pp. 205–227.

[9] Sarven Capadisli et al. "Linked data notifications: a resource-centric communication protocol". In: *European Semantic Web Conference*. Springer. 2017, pp. 537–553.

[10] Gavin Carothers and Eric Prud'hommeaux. *RDF 1.1 Turtle*. W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/2014/REC-turtle-20140225/.

[11] Gavin Carothers and Andy Seaborne. *RDF 1.1 N-Triples*. W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[12] Jeremy Carroll and Graham Klyne. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. W3C, Feb. 2004. URL: https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[13] Wikimedia Commons. *File:Semantic web stack.svg — Wikimedia Commons, the free media repository*. [Online; accessed 19-December-2020]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Semantic_web_stack.svg&oldid=465885386.

[14] Daniele Dell'Aglio et al. "On a Web of Data Streams". In: *ISWC2017 workshop on Decentralizing the Semantic Web*. s.n., Oct. 2017. URL: https://doi.org/10.5167/uzh-149682.

[15] Harm Delva et al. "Geospatial Partitioning of Open Transit Data". In: *International Conference on Web Engineering*. Springer. 2020, pp. 305–320.

[16] Anastasia Dimou et al. "RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data". In: *Proceedings of the 7th Workshop on Linked Data on the Web*. Ed. by Christian Bizer et al. Vol. 1184. CEUR Workshop Proceedings. Apr. 2014. URL: http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[17] Renata Dividino, Thomas Gottron, and Ansgar Scherp. "Strategies for Efficiently Keeping Local Linked Open Data Caches Up-To-Date". In: *SEMANTIC WEB - ISWC 2015, PT II*. Ed. by Arenas, M and Corcho, O and Simperl, E and Strohmaier, M and DAquin, M and Srinivas, K and Groth, P and Dumontier, M and Heflin, J and Thirunarayan, K and Staab, S. Vol. 9367. Lecture Notes in Computer Science. 14th International Semantic Web Conference (ISWC), Bethlehem, PA, OCT 11-15, 2015. Elsevier; Fujitsu; Google; iMinds; Ontotext; Systap; Yahoo Labs; Franz Inc; IBM Res; Oracle; Metaphacts; Blazegraph. 2015, 356–373. ISBN: 978-3-319-25010-6; 978-3-319-25009-0. DOI: {10.1007/978-3-319-25010-6_24}.

[18] Javier Fernández, A. Polleres, and J. Umbrich. "Towards efficient archiving of dynamic linked open data". In: *CEUR Workshop Proceedings* 1377 (Jan. 2015), pp. 34–49.

[19] Javier Fernández et al. "Evaluating query and storage strategies for RDF archives". In: *Semantic Web* 10 (Aug. 2018), pp. 1–45. DOI: 10.3233/SW-180309.

[20] *FlatBuffers*. https://google.github.io/flatbuffers/. Accessed: 2020-11-22.

[21] Pauline Folz, Hala Skaf-Molli, and Pascal Molli. "CyCLaDEs: A Decentralized Cache for Triple Pattern Fragments". In: May 2016, pp. 455–469. ISBN: 978-3-319-34128-6. DOI: 10.1007/978-3-319-34129-3_28.

[22] J. Gregorio and B. de hOra. *The Atom Publishing Protocol*. RFC 5023. RFC Editor, Oct. 2007.

[23] Amy Guy and Sarven Capadisli. *Linked Data Notifications*. W3C Recommendation. W3C, May 2017. URL: https://www.w3.org/TR/2017/REC-ldn-20170502/.

[24] *HDT Binary Format*. https://www.rdfhdt.org/hdt-binary-format/. Accessed: 2020-11-14. July 2015.

[25] Tom Heath and Christian Bizer. "Linked data: Evolving the web into a global data space". In: *Synthesis lectures on the semantic web: theory and technology* 1.1 (2011), pp. 1–136.

[26] Ivan Herman et al. *RDFa 1.1 Primer-Rich Structured Data Markup for Web Documents*. 2015.

[27] *Language Guide*. https://developers.google.com/protocol-buffers/docs/overview. Accessed: 2020-11-14.

[28] Ora Lassila. *Resource Description Framework (RDF) Model and Syntax Specification*. W3C Recommendation. W3C, Feb. 1999. URL: https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[29] Qiao Li et al. "A Novel Cache Replacement Policy for ISP Merged CDN". In: *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*. ICPADS '12. USA: IEEE Computer Society, 2012, pp. 708–709. ISBN: 9780769549033. DOI: 10.1109/ICPADS.2012.106. URL: https://doi.org/10.1109/ICPADS.2012.106.

[30] Frank Manola, Eric Miller, Brian McBride, et al. "RDF primer". In: *W3C recommendation* 10.1-107 (2004), p. 6.

[31] Andrea Mauri et al. "Triplewave: Spreading RDF streams on the web". In: *International Semantic Web Conference*. Springer. 2016, pp. 140–149.

[32] Charles 'chaals' (McCathie) Nevile, Dan Brickley, and Ian Hickson. *HTML Microdata*. W3C Working Draft. https://www.w3.org/TR/2018/WD-microdata-20180426/. W3C, Apr. 2018.

[33] Chifumi Nishioka and Ansgar Scherp. “Keeping linked open data caches up-to-date by predicting the life-time of RDF triples”. In: *Proceedings of the International Conference on Web Intelligence* (2017). DOI: 10.1145/3106426.3106463.

[34] Aaron Parecki and Julien Genestoux. *WebSub*. W3C Recommendation. W3C, Jan. 2018. URL: https://www.w3.org/TR/2018/REC-websub-20180123/.

[35] Danh Phuoc, Josiane Parreira, and Manfred Hauswirth. “Linked Stream Data Processing”. In: Sept. 2012, pp. 245–289. ISBN: 978-3-642-33157-2. DOI: 10.1007/978-3-642-33158-9_7.

[36] Yves Raimond and Guus Schreiber. *RDF 1.1 Primer*. W3C Note. W3C, June 2014. URL: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[37] Julián Andrés Rojas Meléndez et al. “A Preliminary Open Data Publishing Strategy for Live Data in Flanders”. In: *Companion Proceedings of the The Web Conference 2018*. WWW ’18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 1847–1853. ISBN: 9781450356404. DOI: 10.1145/3184558.3191650. URL: https://doi.org/10.1145/3184558.3191650.

[38] Sam Ruby. “Rss 2.0 and atom 1.0 compared”. In: 10 (2008). URL: http://www.intertwingly.net/wiki/pie/Rss20AndAtom10Compared.

[39] Andy Seaborne and Gavin Carothers. *RDF 1.1 TriG*. W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/2014/REC-trig-20140225/.

[40] *SPARQL 1.1 Overview*. W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/.

[41] Manu Sporny et al. “JSON-LD Syntax 1.0”. In: *W3C Community Group Final Specification* (2012).

[42] Michael Stonebraker, Uundefinedur Çetintemel, and Stan Zdonik. “The 8 Requirements of Real-Time Stream Processing”. In: *SIGMOD Rec.* 34.4 (Dec. 2005), pp. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504. URL: https://doi.org/10.1145/1107499.1107504.

[43] Ruben Taelman, Miel Vander Sande, and Ruben Verborgh. “OSTRICH: Versioned Random-Access Triple Store”. In: *Companion Proceedings of the The Web Conference 2018*. WWW ’18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 127–130. ISBN: 9781450356404. DOI: 10.1145/3184558.3186960. URL: https://doi.org/10.1145/3184558.3186960.

[44] Ruben Taelman et al. “Triple storage for random-access versioned querying of RDF archives”. In: *Journal of Web Semantics* 54 (Sept. 2018). DOI: 10.1016/j.websem.2018.08.001.

[45] Jeni Tennison. “CSV on the Web: A Primer”. In: *W3C working group note (World Wide Web Consortium, Cambridge, 2014)* (2016).

[46] Brecht Van de Vyvere, Pieter Colpaert, and Ruben Verborgh. “Comparing a polling and push-based approach for live Open Data interfaces”. In: *International Conference on Web Engineering*. Springer. 2020, pp. 87–101.

[47] Charlie Wang. “HTTP vs. MQTT: A tale of two IoT protocols”. In: (Nov. 2002). DOI: https://cloud.google.com/blog/products/iot-devices/http-vs-mqtt-a-tale-of-two-iot-protocols.

[48] Xiaolin Yi, Fan Zheng, and Wei Jia. “Two Layer Cache for RDF Database”. In: *2013 10TH WEB INFORMATION SYSTEM AND APPLICATION CONFERENCE (WISA 2013)*. 10th Web Information System and Application Conference (WISA), Yangzhou, PEOPLES R CHINA, NOV 01-03, 2013. IEEE Comp Soc; Chinese Comp Federat; IEEE; Yangzhou Univ; Xian Jiaotong Univ; Liaoning Normal Univ; NE Univ; Nanjing Univ. 2013, pp. 11+. ISBN: 978-0-7695-5134-0. DOI: {10.1109/WISA.2013.10}.