Link Traversal Query Processing over Decentralized Environments with Structural Assumptions

Ruben Taelman, Ruben Verborgh

IDLab, Department of Electronics and Information Systems, Ghent University – imec, Ghent, Belgium
ruben.taelman@ugent.be

Abstract. To counter societal and economic problems caused by data silos on the Web, efforts such as Solid strive to reclaim private data by storing it in permissioned documents over a large number of personal vaults across the Web. Building applications on top of such a decentralized Knowledge Graph involves significant technical challenges: centralized aggregation prior to query processing is impossible for legal reasons, and current federated querying techniques cannot handle this large scale of distribution at the expected performance. We propose an extension to Link Traversal Query Processing (LTQP) that incorporates structural properties within decentralized environments to tackle their unprecedented scale. In this article, we analyze the structural properties of the Solid decentralization ecosystem that are relevant for query execution, we introduce novel LTQP algorithms leveraging these structural properties, and evaluate their effectiveness. Our experiments indicate that these new algorithms obtain correct results in the order of seconds, which existing algorithms cannot achieve. This work reveals that a traversal-based querying method using structural assumptions can be effective for large-scale decentralization, but that advances are needed in the area of query planning for LTQP to handle more complex queries. These insights open the door to query-driven decentralized applications, in which declarative queries shield developers from the inherent complexity of a decentralized landscape.

Canonical version: https://comunica.github.io/Article-ISWC2023-SolidQuery/

1 Introduction

Despite transforming our world to be more interconnected than ever before, the Web has become increasingly centralized in recent years, contrary to its original vision [1]. The majority of data on the Web today is flowing towards *data silos*, which are in the hands of large companies. This siloization of data leads to various problems, ranging from issues with cross-silo interoperability and vendor lock-in, to privacy breaches and individuals' data being controlled by companies instead of themselves.
Because of increasing awareness and user-empowering legislation such as the GDPR and CCPA, decentralization initiatives [2, 3, 4, 5] are gaining popularity. Their common goal is to give people back control over their own data by guarding it in chosen locations on the Web instead of aggregated in silos. Initiatives such as Solid [2] and Bluesky [3] achieve this by allowing users to store any kind of data in their own *personal data vault*, which they fully control. In Solid, these data vaults form personal

Knowledge Graphs [6, 7], represented as collections of Linked Data documents [8] containing RDF triples [9]. The presence of such data vaults results in a large-scale distribution of data, where applications involving multiple individuals' data will require accessing thousands or even millions of access-controlled data items in data vaults across the Web.

The state of the art in RDF query processing does not support such applications, as it is unprepared to handle large-scale decentralization. The RDF querying research of the past two decades has focused on data that either *is* or *can be* centralized. Some datasets would be in a private or public quadstore [10], specifically designed to enable efficient query processing. Other datasets would be Linked Open Data with a permissive license, making it possible to copy or cache data in a local quadstore. This permanent option to centralize limited the need for decentralized query techniques that allow data to remain at its source. After all, downloading and aggregating datasets would nearly always be more efficient. Yet when we use Linked Data for permissioned or sensitive data, it becomes impossible to download datasets in their entirety.

While *federated* query execution approaches do exist [11, 12, 13, 14], they currently handle a *small* number (~10) of *large* sources, whereas decentralized environments such as Solid are characterized by a *large* number ($>10^3$) of *small* sources. Furthermore, federated query execution techniques assume sources to be known prior to query execution, which is not feasible in decentralized environments due to the lack of a central index. Hence, existing techniques are ill-suited for the envisaged scale of decentralization.

Link Traversal Query Processing (LTQP) [15, 16] is an alternative query execution paradigm that is more promising for uncharted decentralized environments. LTQP allows querying over a continuously growing selection of source documents that are discovered *during* query execution, by following hyperlinks between Linked Data documents using the *follow-your-nose* principle [8]. Although LTQP has been theoretically interesting, it has not seen any practical use so far, in particular because of performance concerns.

Fortunately, decentralized ecosystems such as Solid exhibit additional structural properties in addition to the Linked Data principles, allowing us to improve query performance through assumptions about data, documents, and their organization. For example, Solid makes use of Linked Data Platform [17] containers to provide *completeness guarantees* when finding data within vaults, and provides a Type Index [18] to enable *type-based document discovery*.

In this work, we prove that LTQP can be an effective paradigm for the structured retrieval of data from unindexed decentralized Knowledge Graphs. The key is to exploit the aforementioned structural properties for more effective source discovery and query optimization. While we apply our research to the Solid ecosystem, these concepts may be generalizable to other decentralization initiatives [3, 4, 5].

This article is structured as follows. In the next section, we discuss related work, after which we provide an analysis of the structural properties of Solid data vaults in Section 3. In Section 4, we introduce LTQP algorithms that make use of these structural properties, which are evaluated in Section 5. Finally, we conclude in Section 6.

2 Related Work

The Link Traversal Query Processing (LTQP) paradigm was introduced more than a decade ago [19] as a way to query over the Web of Linked Open Data as if it was a globally distributed dataspace, without having to first index it in a single location. LTQP does this by employing the *follow-your-nose* principle of Linked Data [8] during query execution, in which new RDF are continuously added to a local dataset while discovering new sources by following links between documents. An iterator-based pipeline [19] allows execution to take place without waiting until all links have been followed.

In contrast to *two-phase* approaches [20, 21] that perform data retrieval and indexing *before* query execution, LTQP is thus an *integrated* approach [22] with *parallel* source discovery and query execution. As a one-phase approach, LTQP cannot rely on traditional pre-execution optimization algorithms that require prior dataset statistics. A zero-knowledge query planning technique [23] instead orders triple patterns in a query based on link traversal-specific heuristics. LTQP is related to the idea of *SQL-based query execution over the Web* [24, 25] and to the concept of *focused crawling* [26, 27]. While LTQP considers the Web of Linked Open Data a large database using the RDF data model, SQL-based approaches focus on querying attributes or content within Web pages. Focused crawlers search for Web pages of specific topics to populate a local database or index, using a two-phase approach where a preprocessing step precedes execution. While two-phase approaches in general are able to produce better query plans using traditional cardinality-based planning techniques, waiting for data retrieval to be completed may be impractical or even impossible for certain queries.

The number of potential links to be followed within the Web of Linked Open Data can become prohibitively large. In the worst case, a single query could theoretically require traversing the entire Web. Therefore, the formal LTQP model [16] enables different *reachability criteria*, which embody strategies for deciding what links to follow, each leading to different result completeness semantics. *cNone* follows no URLs, *cAll* follows all URLs in all encountered triple components, and *cMatch* only follows URLs in triple components for those triples that match a triple pattern within the query. *Context-based semantics* [28] is an extension of these reachability semantics to cope with property path expressions in the SPARQL 1.1 language [29]. Next to query-driven reachability, another extension [30] introduces the ability for data publishers to express which links should be followed using *subweb specifications*.

In addition to filtering links via different semantics, a second methodology for lowering query result arrival times is through *link prioritization* [31]. However, existing techniques only sometimes result in faster query results compared to no prioritization.

Even though multiple query languages [32, 33, 34] have been introduced specifically for LTQP, its SPARQL-based execution model [19] is still the most widely used. Since SPARQL is the only language among these that is a standard, and the fact that it is more widely known and supported by different tools, we make use of it within this work. Nevertheless, the concepts within this work can be applied to other languages.

In general, LTQP has mostly been applied to querying Linked *Open* Data on the Web. In contrast, our work applies LTQP to decentralized environments with *private* personal data, organized in personal data vaults with specific structural properties.

3 The Solid Ecosystem

In this section, we provide an analysis of the structural properties within the Solid ecosystem that are relevant for query processing. We start by explaining the concept of data vaults and their implications on applications. Next, we explain the WebID, which is used for identifying users. Finally, we discuss the Solid type index; a structural property that improves data discovery.

3.1 Data Vault

The Solid Protocol [35] supports the concept of a personal data vault (also known as *data pod*), which is a user-controlled space in which any kind of public and private data can be stored. Users can choose where and how their vault is stored on the Web, by hosting it themselves [36], or obtaining service-provided space by a company [37] or government [38]. Data vaults are intended to be loosely coupled to applications, and applications must request explicit access to the user for interacting with specific data. This loose coupling enables different applications to use the same data in an interoperable way.
Current data vaults are primarily document-oriented, and are exposed on the Web as a REST API using elements of the Linked Data Platform (LDP) specification [17]. Directories are represented using *LDP Basic Containers*, which can contain any number of resources that correspond to RDF or non-RDF resources (via `ldp:contains` links), or other nested basic containers. For the remainder of this article, we will only consider the processing of RDF resources and containers within vaults. Resources within vaults can be read by sending HTTP `GET` requests to their URLs, with optional content negotiation to return the documents in different RDF serializations. Vaults may also support creation and modification using HTTP `PATCH` and `POST` requests. An example of such a basic container can be found in Listing 1.

```
PREFIX ldp: <http://www.w3.org/ns/ldp#>
<> a ldp:Container, ldp:BasicContainer, ldp:Resource;
   ldp:contains <file.ttl>, <posts/>, <profile/>.
<file.ttl> a ldp:Resource.
<posts/> a ldp:Container, ldp:BasicContainer, ldp:Resource.
<profile/> a ldp:Container, ldp:BasicContainer, ldp:Resource.
```

Listing 1: An LDP container in a Solid data vault containing one file and two directories in the Turtle serialization.

Data vaults can contain public as well as private data. Users can configure who can access or modify files within their vault using mechanisms such as ACL [39] and ACP [40]. This configuration is usually done by referring to the *WebID* of users.

3.2 WebID Profile

Any agent (person or organization) within the Solid ecosystem can establish their identity through a URI, called a *WebID*. These agents can authenticate themselves using the decentralized Solid OIDC protocol [41], which is required for authorizing access during the reading and writing of resources. Each WebID URI should be dereferenceable, and return a WebID profile document. Next to basic information of the agent such as its name, this document contains links to 1) the vault's LDP container (via `pim:storage`), and 2) public and private type indexes. An example is shown in Listing 2.

```
PREFIX pim: <http://www.w3.org/ns/pim/space#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX solid: <http://www.w3.org/ns/solid/terms#>
<#me> foaf:name "Zulma";
      pim:storage </>;
      solid:oidcIssuer <https://solidcommunity.net/>;
      solid:publicTypeIndex </publicTypeIndex.ttl>.
```

Listing 2: A simplified WebID profile in Turtle.

3.3 Type Index

Users are free to organize documents in their vault as they see fit. The Type Index [18] is a document that enables type-based resource discovery within a vault. Users may have public or private type indexes, which respectively refer to data that are and are not publicly discoverable. A type index can contain type registration entries for different classes, where each registration has a link to resources containing instances of the corresponding class. Listing 3 shows a type index example with type registrations for posts and comments, where the posts entry refers to a single posts file, and the comments entry refers to a container with multiple comments files. If an application wants to obtain all posts of a user, it can follow the link within the type index entry corresponding to the *post* class.

```
PREFIX ldp: <http://www.w3.org/ns/ldp#>
<> a solid:TypeIndex ;
   a solid:ListedDocument.
<#ab09fd> a solid:TypeRegistration;
  solid:forClass <http://example.org/Post>;
  solid:instance </public/posts.ttl>.
<#bq1r5e> a solid:TypeRegistration;
  solid:forClass <http://example.org/Comment>;
  solid:instanceContainer </public/comments/>.
```

Listing 3: Example of a type index with entries for posts and comments in Turtle.

4 Approach

In this section, we introduce techniques for handling Solid's structural properties discussed in Section 3. We do not introduce any additional components or structural properties to the Solid ecosystem; instead, we use what Solid vaults already provide today, and investigate how to query over them as efficiently as possible. We start by discussing the preliminaries of the formalizations we will introduce. Next, we discuss our pipeline-based link queue approach. Then, we discuss two novel discovery approaches for LTQP. Finally, we discuss their implementations.

4.1 Formal preliminaries

Hereafter, we summarize the semantics of SPARQL querying [42] and LTQP [16, 30]. The infinite set of *RDF triples* is formalized as $\mathcal{T} = (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, where $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ respectively denote the disjoint, infinite sets of *IRIs*, *blank nodes*, and *literals*. Furthermore, $\mathcal{V}$ is the infinite set of all variables that is disjoint from $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$. A tuple $tp \in (\mathcal{V} \cup \mathcal{I}) \times (\mathcal{V} \cup \mathcal{I}) \times (\mathcal{V} \cup \mathcal{I} \cup \mathcal{L})$ is called a *triple pattern*. A finite set of these triple patterns is called a *basic graph pattern* (BGP). For the formalization, we only consider BGPs since they form the foundational building block of a SPARQL query; our implementation incorporates all of SPARQL 1.1. The query results of a SPARQL query P over a set of RDF triples G are called *solution mappings*, which are denoted by $[[P]]_G$, consisting of partial mappings $\mu : \mathcal{V} \to (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$. An RDF triple t *matches* a triple pattern tp if $\exists \mu : t = \mu[tp]$, where $\mu[tp]$ is the triple pattern that is obtained by replacing all variables from μ in tp.
Formally, the reachability approaches that were discussed in Section 2 define which links should be followed during link traversal, and are usually captured as *reachability criteria* [16]. Since these reachability semantics lack expressive power to capture the structural properties we require, we formalize new reachability criteria in this work as *source selectors* within the subweb specification formalization [30]. Within this formalization, a source selector σ is defined as $\sigma : \mathcal{W} \to 2^{\mathcal{I}}$, where $\mathcal{W}$ is a Web of Linked Data. The Web of Linked Data $\mathcal{W}$ is a tuple $\langle D, data, adoc \rangle$, where D is a set of available documents, $data$ a function from D to $2^{\mathcal{T}}$ associating each document with its contained triples, and $adoc$ a partial function from $\mathcal{I}$ to D to dereference documents.
We define the set of all Solid data vaults as Υ, where each vault $v \in \Upsilon$ is defined by its set of triples [7], where $triples(v) \subseteq \mathcal{T}$. For a vault v_{LDP} exposed through the LDP interface, the triples contained in such a vault are captured in different documents $D_v \subseteq D$. Hereby, $triples(v_{LDP}) = \cup_{d \in D_v} data(d)$.

4.2 Pipeline-based link queue

To execute a query, our approach builds upon the zero-knowledge query planning technique [23] to construct a logical query plan ahead of query execution. This resulting plan produces a tree of logical query operators representing the query execution order. To execute this plan, the logical operators are executed by specific physical op-

erators. Our physical query execution builds upon the iterator-based pipeline approach [19], which is the most popular among LTQP implementations [43, 44, 45]. We consider the execution plan as a pipeline [46] of iterator-based physical operators, where intermediary results flow through chained operators with pull-based results.
Instead of letting operators trigger the dereferencing of URIs [19], we follow a link queue-based approach [31]. The architecture of this approach is visualized in Fig. 1. Concretely, we consider a continuously growing *triple source* as the basis of the pipeline tree, which is able to produce a (possibly infinite) stream of RDF triples. This triple source is fed triples originating from a loop consisting of the *link queue*, *dereferencer*, and a set of *link extractors*. The link queue accepts links from a set of link extraction components, which are invoked for every document that has been dereferenced by the dereferencer. The dereferenced documents containing triples are also sent to the continuously growing triple source. This link queue is initialized with a set of seed URIs, and the dereferencer continuously dereferences the URIs in the queue until it is empty. Since the link extractors are invoked after every dereference operation, this queue may virtually become infinitely long.

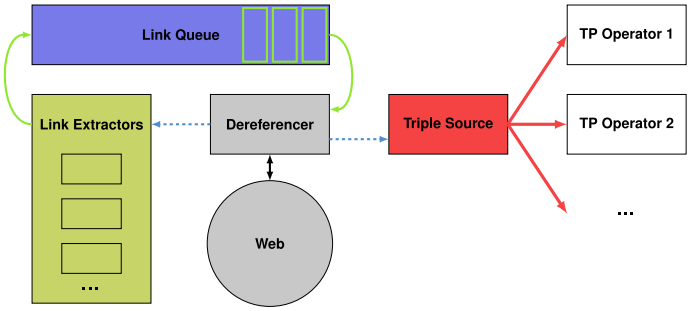


Fig. 1: Link queue, dereferencer, and link extractors feeding triples into a triple source, producing triples to tuple-producing operators in a pipelined query execution.

This link queue and link extractor approach is generic enough to implement other LTQP methods [19, 16, 28, 30, 31] for determining and prioritizing links that need to be followed. For example, one extractor may consider `rdfs:seeAlso` links, while another extractor may consider URIs of a triple that matches with a triple pattern from the query. Optionally, operators in the query pipeline may push links into the link queue, which enables context-based reachability semantics [28]. Link extractors only consider URIs as links, and ignore matching blank nodes and literals.
The triple source is connected to all tuple-producing SPARQL operators [42] in the leaves of the query plan, such as triple patterns and property path operators, into which a stream of triples is sent. The source indexes all triples, to ensure that an operator that is executed later in the execution process does not miss any triples.

4.3 Discovery of data vault

Below, we introduce a novel discovery approach for traversing over Solid data vaults.

Intuitive description To achieve link traversal within a vault, we assume that the WebID document is available as seed URI, or is discovered through some other reachability approach. As discussed in Section 3, the root of a vault can be discovered from a WebID document by dereferencing the object URI referred to by the `pim:storage` predicate. Next, all resources within this vault can be discovered by recursively following `ldp:contains` links from the root container.
We only consider triples for the `pim:storage` and `ldp:contains` predicates that have the current document URI as subject. If subjects contain fragment identifiers, we only consider them if the current document URI had this fragment identifier as well before it was dereferenced. For example, if a WebID with fragment identifier `#me` was discovered, then we only consider triples with that full WebID as subject.
Formal description We can formalize our discovery approach for the roots of data vaults as a following source selector starting from a given WebID with URI i as $\sigma_{\text{SolidVault}}(W) = \{o \mid \langle i \text{ pim:storage } o\rangle \in data(adoc(i))\}$. Disjunctively coupled with this, we can formalize a source selector that can recursively traverse an LDP container as $\sigma_{\text{LdpContainer}}(W) = \{o \mid \forall s : \langle s \text{ ldp:contains } o\rangle \in data(adoc(s)\}$

4.4 Discovery of type index

As discussed in Section 3, the type index enables class-based resource discovery in a vault. In this section, we introduce a novel method that follows links in the type index, with an optional filter that only follows links matching with a class in the query.
Intuitive description As before, we consider a WebID document as the starting point. From this document, we follow the `solid:publicTypeIndex` and `solid:privateTypeIndex` links. For each discovered type index, we consider all `solid:TypeRegistration` resources, and follow their `solid:instance` and `solid:instanceContainer` links.
As an optimization, we can also take into account the type information within the registrations of the type index, to only follow those links for classes that are of interest to the current query. Concretely, this involves considering the objects referred to by `solid:forClass` on each type registration. To know whether or not a class is relevant to the current query, we explicitly check for the occurrence of this class within the query as object within triples using the `rdf:type` predicate. For subjects in the query without `rdf:type` predicate, the matching class is unknown, which is why we consider all type registrations in this case.
Formal description To discover and traverse type indexes, we formalize the following source selector from a given WebID with URI s when querying a BGP B:

$$
\begin{aligned}
&\sigma_{\text{SolidTypeIndex}}(W) = \{o \mid \forall t, r, c : \phi(B, c)\\
\wedge\ &(\langle s \text{ solid:publicTypeIndex } t\rangle \in data(adoc(s))\\
&\vee\langle s \text{ solid:privateTypeIndex } t\rangle \in data(adoc(s)))\\
\wedge\ &(\langle r \text{ rdf:type solid:TypeRegistration}\rangle \in data(adoc(t))\\
&\wedge\langle r \text{ solid:forClass } c\rangle \in data(adoc(t)))\\
\wedge\ &(\langle r \text{ solid:instance } o\rangle \in data(adoc(t))\\
&\vee\langle r \text{ solid:instanceContainer } o\rangle \in data(adoc(t)))\}
\end{aligned}
$$

Since `solid:instanceContainer` links to LDP containers, $\sigma_{\text{SolidTypeIndex}}$ should be disjunctively combined with $\sigma_{\text{LdpContainer}}$.
In this formalization, we consider $\phi(B, c)$ a filtering predicate function for determining which classes are considered within the type index. To consider *all* type registrations within the type index, we can implement $\phi(B, c)$ as a predicate always returning `true`. To only consider type registrations that match with a class mentioned in the query, we introduce the following filtering function:

$$\phi_{\text{QueryClass}}(B, c) = \begin{cases} \text{true} & \text{if } \exists tp \in B : \\ & \langle ?v \text{ rdf:type } c \rangle \text{ matches } tp \\ & \text{or if } \exists s : \langle s \; ?p \; ?o \rangle \in B \\ & \wedge \{o \mid \langle s \text{ rdf:type } o \in B\} = \emptyset \\ \text{false} & \text{else.} \end{cases}$$

4.5 Implementation

We have implemented our system as new components for the Comunica SPARQL framework [47]. Concretely, we implemented the pipeline-based link queue as a separate module, and we provide link extractors corresponding to the source selectors introduced in previous sections. We fully support SPARQL 1.1, and have pipelined implementations of all monotonic SPARQL operators. Pipelining is important for iterative tuple processing in a non-blocking manner, as the link queue and the resulting stream of triples may become infinitely long.
Our implementation focuses on the SPARQL query language, instead of alternatives such as LDQL [33] and SPARQL-LD [48] that incorporate link navigation paths into the query. As discussed in Section 3, different Solid apps or user preferences may lead to the storage of similar data at different locations within vaults. Hence, link navigation must be *decoupled* from the query to make queries reusable for different Solid users, as link paths to data may differ across different vaults. Our implementation uses LDP container traversal and the type index to replace explicit navigation links.
To provide a stable reference implementation that can be used for the experiments in this work and future research, our implementation focuses on extensibility and reusability. Our implementation builds upon best practices in LTQP and lessons learned from other implementations [43] including, the use of client-side caching [49], the different reachability semantics [16], zero-knowledge query planning [23] applied to arbitrary join operations instead of only triple patterns in BGPs, and more [19]. Furthermore, our implementation allows users to explicitly pass seed URIs, but falls back to query-based seed URIs [43] if no seeds were provided. This fallback considers all URIs within the query as seed URIs.
As Solid performs access control at document-level, we enable users to authenticate to the client-side query engine. This allows the query engine to perform authenticated requests on behalf of the user. Since authenticated requests happen purely on the HTTP layer, other parts of the query engine do not have to be concerned about authentication, and the processing of public and private data can happen together transparently in the client-side query engine.

As a result, our implementation can query over one or more Solid data vaults. To ensure that common HTTP errors that may occur during link traversal don't terminate the query execution process, we enable a default *lenient* mode, which ignores dereference responses with HTTP status code in ranges 400 and 500.

5 Evaluation

In this section, we tackle the research question *"How well does link traversal query processing perform over decentralized environments with structural properties?"*. Within this work, we apply our experiments to the structural properties of the decentralized environment provided by Solid, but findings may be generalizable to other decentralized environments. We provide an answer to this research question by evaluating different approaches based on the implementation discussed in Section 4, using a benchmark that simulates Solid data vaults. We first introduce the design of our experiment, followed by presenting our experimental results, and a discussion of our results to answer our research question.

5.1 Experimental Design

Our experimental design is based on the SolidBench benchmark that simulates a realistic decentralized environment based on the Solid ecosystem. Concretely, the benchmark generates a configurable number of data vaults with configurable sizes containing social networking data, where a variety of fragmentation strategies are used to organize files in vaults. By default, it generates 158.233 RDF files over 1.531 data vaults with a total of 3.556.159 triples across all files. Furthermore, it provides SPARQL query templates that simulate a realistic workload for a social networking application. The underlying dataset and query templates are derived from the Social Network Benchmark (SNB) [50]. A full description of all queries can be found on https://github.com/SolidBench/SolidBench.js/blob/master/templates/queries/README.md.

We make use of a factorial experiment containing the following factors and values:

- **Vault discovery combinations**: None, LDP, Type Index, Filtered Type Index, LDP + Type Index, LDP + Filtered Type Index
- **Reachability semantics**: cNone, cMatch, cAll

The LDP strategy corresponds to the disjunction of the source selectors $\sigma_{\text{SolidVault}}$ and $\sigma_{\text{LdpContainer}}$, the Type Index to $\sigma_{\text{LdpContainer}}$ and $\sigma_{\text{SolidTypeIndex}}$ with $\phi(B, c)$ always returning `true`, and the Filtered Type Index to $\sigma_{\text{LdpContainer}}$ and $\sigma_{\text{SolidTypeIndex}}$ with $\phi_{\text{QueryClass}}$.

Our experiments were performed on a 64-bit Ubuntu 14.04 machine with a 24-core 2.40 GHz CPU and 128 GB of RAM. The Solid vaults and query client were executed in isolated Docker containers on dedicated CPU cores with a simulated network. All queries were configured with a timeout of two minutes, and were executed three times

to average metrics over. Each query template in the benchmark was instantiated five times, resulting in 40 discover queries, 35 short queries, and 60 complex queries. These query templates are available in the supplementary material.
We were unable to compare our implementation to existing LTQP engines, because those systems (e.g. Lidaq [21]) would either require significant changes to work over Solid vaults, they depend on a non-standard usage of the SPARQL syntax (e.g. SPARQL-LD [48]), or insufficient documentation was present to make them work (e.g. SQUIN [43]). Nevertheless, in order to ensure a fair and complete comparison, we have re-implemented the foundational LTQP algorithms (cNone, cMatch, cAll), and compare them against, and in combination with, our algorithms.

5.2 Experimental Results

In this section, we present results that offer insights into our research question. Table 1 and Table 2 show the aggregated results for the different combinations of our setup for the discover and short queries of the benchmark, respectively. We omit results from complex queries, as none of the approaches achieve a level of accuracy significantly higher than 0%. Concretely, each table shows the average ($\overline{t}$) and median ($\tilde{t}$) execution times (ms), the average ($\overline{t}_1$) and median ($\tilde{t}_1$) time until first result (ms), average number of HTTP requests per query ($\overline{req}$), total number of results on average per query ($\sum ans$), average accuracy ($\overline{acc}$), and number of timeouts ($\sum to$) across all queries. The combinations with the highest accuracy value are marked in bold. The number of HTTP requests is counted across all query executions that did not time out within each combination. The timeout column represents the number of query templates that lead to a timeout for a given combination. The accuracy of each query execution is a percentage indicating the precision and recall of query results to the expected results.

	$\overline{t}$	$\tilde{t}$	$\overline{t}_1$	$\tilde{t}_1$	$\overline{req}$	$\sum ans$	$\overline{acc}$	$\sum to$
cnone-base	40	0	N/A	N/A	8	0.00	0.00%	0
cmatch-base	1,791	0	22,946	24,439	1,275	0.00	0.00%	1
call-base	128,320	127,021	28,448	10,554	0	0.63	3.13%	8
cnone-idx	1,448	842	447	351	243	20.50	74.14%	0
cmatch-idx	**12,284**	**2,210**	**2,304**	**1,217**	**2,567**	**39.13**	**99.14%**	**0**
call-idx	124,197	124,811	48,223	9,778	18,022	3.13	17.40%	7
cnone-idx-filt	1,429	755	435	311	230	20.50	74.14%	0
cmatch-idx-filt	**12,114**	**2,312**	**2,397**	**1,075**	**2,554**	**39.13**	**99.14%**	**0**
call-idx-filt	124,003	126,093	43,147	29,937	11,023	4.50	29.78%	8
cnone-ldp	1,606	994	563	386	342	20.50	74.14%	0
cmatch-ldp	13,463	2,288	3,660	1,057	3,625	37.88	86.64%	1
call-ldp	123,712	123,479	37,083	13,733	0	2.00	16.25%	8
cnone-ldp-idx	1,560	1,001	482	349	358	20.50	74.14%	0
cmatch-ldp-idx	**12,417**	**2,529**	**2,333**	**1,189**	**2,709**	**39.13**	**99.14%**	**0**
call-ldp-idx	127,768	125,103	67,577	13,472	12,466	2.38	16.63%	7
cnone-ldp-idx-filt	1,552	1,006	425	331	357	20.50	74.14%	0
cmatch-ldp-idx-filt	**12,483**	**2,372**	**2,309**	**925**	**2,708**	**39.13**	**99.14%**	**0**
call-ldp-idx-filt	123,979	125,235	48,382	10,368	16,623	3.13	17.40%	7

Table 1: Aggregated results for the different combinations across 8 **discover** queries.

	$\bar{t}$	$\tilde{t}$	$\bar{t}_1$	$\tilde{t}_1$	$\overline{req}$	$\sum ans$	$\overline{acc}$	$\sum to$
cnone-base	34,364	70	18	2	12	0.14	14.29%	2
cmatch-base	47,700	987	121	92	592	0.43	42.86%	3
call-base	126,794	125,609	1,547	787	0	0.00	0.00%	7
cnone-idx	34,775	540	676	151	71	0.14	14.29%	2
cmatch-idx	70,142	119,114	6,837	530	263	0.43	42.86%	4
call-idx	109,943	123,227	14,290	19,345	0	0.00	0.00%	7
cnone-idx-filt	34,804	534	527	110	71	0.14	14.29%	2
cmatch-idx-filt	69,808	119,032	7,190	434	263	0.43	42.86%	4
call-idx-filt	116,618	123,312	9,764	6,207	0	0.00	0.00%	7
cnone-ldp	34,975	621	816	46	96	0.29	15.71%	2
cmatch-ldp	**70,026**	**119,586**	**6,524**	**636**	**291**	**0.57**	**44.29%**	**4**
call-ldp	127,550	126,587	717	483	0	0.00	0.00%	7
cnone-ldp-idx	34,852	811	521	43	100	0.14	14.29%	2
cmatch-ldp-idx	69,534	119,215	2,936	437	295	0.43	42.86%	4
call-ldp-idx	110,217	122,525	8,841	6,114	0	0.00	0.00%	7
cnone-ldp-idx-filt	34,830	742	402	83	100	0.14	14.29%	2
cmatch-ldp-idx-filt	**70,042**	**119,126**	**6,246**	**663**	**295**	**0.57**	**44.29%**	**4**
call-ldp-idx-filt	114,800	123,058	15,075	17,192	0	0.00	0.00%	7

Table 2: Aggregated results for the different combinations across 7 **short** queries.

These results show that there are combinations of approaches that achieve a very high level of accuracy for discover queries, and a medium level of accuracy for short queries. We will elaborate on these results in more detail hereafter.

5.3 Discussion

Intra-vault and inter-vault data discovery The results above show that if we desire accurate results, that the combination of cMatch semantics together with at least one of the data vault discovery methods is required. This combination is needed because our workload contains queries that either target data within a single vault (e.g. D1), or data spanning multiple data vaults (e.g. D8). While the different data vault discovery methods are able to discover data *within* vaults, the reachability of cMatch is required to discover data *across* multiple vaults.

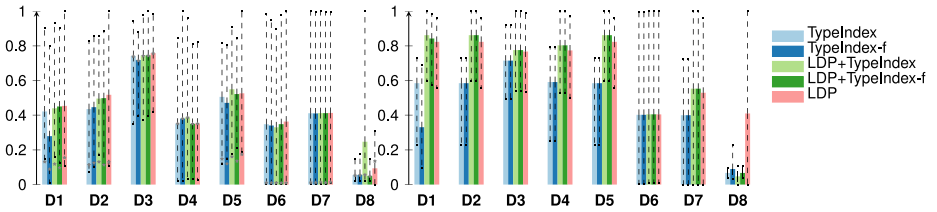
Due to this, cNone (follow no links) is an ineffective replacement for cMatch (follow links matching query) even when combined with discovery methods, because link traversal across multiple vaults will not take place, which will lead to too few query results. Concretely, for discover queries cNone can only achieve a accuracy of 74.14% for discover queries and 28.57% for short queries, compared to respectively 99.14% and 42.86% for cMatch. However, for those queries that target a single vault, cNone can be used instead of cMatch without a loss of accuracy, leading to a lower number of HTTP requests and lower query execution times.

Since cAll leads to all links being followed, including those followed by cMatch, it is theoretically a sufficient replacement for cMatch. However, our results show that cAll follows too many links, which leads to timeouts for nearly all queries.

Our results show that solely using reachability semantics (cMatch or cAll) without a data discovery method is insufficient for discover queries, where a accuracy of only up to 3.13% can be achieved for discover queries. However, when looking at the short queries category, solely using reachability semantics appears to be sufficient, with the query execution time even being lower. This difference exists because the discover workload contains queries that discover data related to a certain person or resource,

while the short queries target only details of specific resources. Discover queries therefore depend on an overview of the vault, while short queries only depend on specific links between resources within a vault. The remainder of this discussion only focuses on discover queries, since these achieve the highest level of accuracy. As such, the short and complex queries highlight opportunities for future improvement.

Type index and LDP discovery perform similarly When comparing the number of HTTP requests and query execution times for different data vault discovery approaches under cMatch in Table 1, we can observe that using the type index leads to fewer HTTP requests and faster query execution compared to LDP-based discovery on average. To explain this behaviour in more detail, Subfig. 2.1 shows the average query execution times of each discover query separately, for the different combinations of data vault discovery approaches. To simplify comparability, the execution times within this figure are relative to the maximum query execution time per query [31]. Furthermore, Subfig. 2.2 shows the average number of HTTP requests for each of those discover queries, which are also made relative to the maximum number of requests per query for better comparability.



Subfig. 2.1: Execution times. **Subfig. 2.2:** Number of HTTP requests.

Fig. 2: Relative measurements for discover queries with different discovery methods under cMatch. Bars indicate average values, whiskers indicate the maxima and minima, and stars indicate average time until first result.

While Subfig. 2.1 shows that for all queries using just the type index is slightly faster or comparable to just LDP-based discovery, this difference has no statistical significance ($p = 0.40$). However, Subfig. 2.2 shows that the number of HTTP requests with the type index is always significantly lower than via LDP ($p = 0.01$).

When the filter-enabled type index approach is used, five queries (D1, D3, D5, D6, D7) are made even faster compared to the non-filtered type index approach. This is because those queries target a possibly empty subset of the type index entries, which means that a significant range of links can be pruned out, which leads to a major reduction in the number of HTTP requests, which is a main bottleneck in link traversal. For the other queries, the filter-enabled approach becomes slightly slower than (D2, D4) or is comparable to (D8) the non-filtered type index approach. For those queries, the processing overhead of type index filtering becomes too high compared to its potential benefit. Statistically, this difference has no significance in terms of execution time ($p = 0.69$) and number of HTTP requests ($p = 0.68$).

These results show that using the type index together with LDP-based discovery is not *significantly* better than the other approaches ($p = 0.71$), which is primarily caused by the statistically significantly higher number of HTTP requests ($p = 0.02$) required for traversing both the type index and nested LDP containers. Query D8 does however show that this combination deserves further investigation, because this query has a result limit that leads to a prioritization of links via the type index, leading to earlier query termination with fewer requests.

In general, results hint that the LDP-based approach combined with filtered type index approach performs better than the other approaches, but this difference is too minor to be significant, hence all approaches can be considered equivalent.

Zero-knowledge query planning is ineffective While it may seem obvious to assume that higher query execution times are caused by a higher number of links that need to be dereferenced, we observe only a weak correlation ($\rho = 0.32$) of this within the cMatch-based discovery approaches discussed before. The main bottleneck in this case appears not primarily to be the number of links to traverse. Instead, our analysis suggests that query plan efficiency is the primary influencer of execution times. This is in contrast to earlier research over Linked Open Data [31], where we instead consider structural properties with more selective link traversal.

To empirically prove this finding, we compare the execution times of our default integrated query execution approach (cMatch with filtered type index discovery) with a two-phase query execution approach that we implemented in the same query engine. Instead of following links during query execution as in the integrated approach, the two-phase approach first follows links at the same rate to index all discovered triples, and processes the query in the traditional *optimize-then-execute* manner. This two-phase approach is based on an oracle that provides all query-relevant links, which we determined by analyzing the request logs during the execution of the integrated approach. Therefore, this two-phase approach is merely a theoretical case, which delays time until first results due to prior indexing, and may not always be achievable due to infinitely growing link queues for some queries. The results of this are in Fig. 3.

Query	Integrated	Two-phase	HTTP Requests
D1	1,077.58	403.54	222
D2	1,020.67	567.57	223
D3	1,193.01	821.23	429
D4	3,266.62	505.00	228
D5	522.23	387.24	223
D6	710.16	289.72	122
D7	626.96	340.54	122
D8	2,037.85	1,654.02	420

Fig. 3: Integrated and two-phase execution times (ms) of discover queries, with number of HTTP requests per query.

These results show that the two-phase approach is on average two times faster for all queries compared to the integrated approach, even when taking into account time for dereferencing. The reason for this is that the two-phase approach is able to perform traditional query planning [51, 52], since it has access to an indexed triple store with planning-relevant information such as cardinality estimates. Since the integrated ap-

proach finds new triples *during* query execution, it is unable to use them for traditional planning. Instead, our integrated approach makes use of the zero-knowledge query planning technique [23] that uses heuristics to plan the query before execution.
Since the only difference between the integrated and two-phase implementations is in how they plan the query, we can derive that the query plan of the integrated approach is ineffective. Hence, there is a need for better query planning during integrated execution, where performance could ideally become more than two times better.
Zero-knowledge query planning [23] is ineffective in our experiments because it has been designed under the assumptions of Linked Open Data, while it does not match with the structural assumptions of specific decentralized environments such as Solid. For example, one of the heuristics within this planner deprioritizes triple patterns with vocabulary terms, such as `rdf:type`, since they are usually the least selective. However, when a Solid type index is present, such types may instead become *very selective*, which means that those would benefit from prioritization. As such, there is a need for alternative query planners that consider the structural assumptions within specific decentralized environments.

6 Conclusions

User-oriented decentralized applications require results in the order of seconds or less to avoid losing the user's attention [53]. Our work has shown that Link Traversal Query Processing is able to achieve such timings, especially as it is able to produce results in an iterative manner, with first results mostly being produced in less than a second. As such, LTQP with the algorithms introduced in this work is effective for querying over decentralized environments with specific structural properties, but there are open research opportunities for optimizing more complex queries as provided by the benchmark. We have shown this by applying LTQP to simulated Solid environments, for which we have introduced algorithms to capture these structural properties.
Before this work, LTQP tended to exclusively focus on Linked Open Data, which centralized quadstores handle better anyway. After all, in addition to major performance concerns, it was assumed that *"we should never expect complete results"* [19] with LTQP because of the unbounded and distributed nature of the Web. Prior aggregation was therefore always the logical option for efficient queries, limiting the evaluation space to fixed datasets with a compatible license.
However, LTQP becomes relevant again within decentralized environments such as Solid, where aggregation is not an option because of permissioning. Performance concerns can be addressed through the usage of additional assumptions during query execution. For instance, the ability to close the world around Solid vaults, and the data discovery techniques that Solid vaults provide, create opportunities for query execution that allow us to guarantee complete results. While we have investigated the specific case of querying Solid vaults, these concepts may be generalizable to other decentralization efforts [3, 4]. This is possible, because our approach solely relies on the structural properties provided by specifications such as LDP [17] and the Type Index [18], which can be used outside of the Solid ecosystem.

LTQP research over Linked Open Data is exploring the direction of finding query-relevant documents as early as possible [31] due to the possibility of incomplete results. In the context of Solid, we have shown that finding all query-relevant documents is not the main bottleneck during query execution anymore. Instead, the *effectiveness of the query plan* has become the new bottleneck. While finding query-relevant documents is still relevant for specific decentralized environments, we show the need for more research towards better query planning techniques. Since LTQP leads to data being discovered during query execution, adaptive query planning [54] techniques are highly promising. So far, these techniques have only seen limited adoption within LTQP [31] and SPARQL query processing [55, 56, 57].

Our findings indicate that discovery approaches such as the Solid Type Index harbor a great potential for improving query performance, and future work in the direction of *implicit type knowledge* within queries (e.g., through RDFS reasoning) and query decomposition over *different type index entries* could be relevant. Furthermore, alternative structural properties could offer more expressivity, such as *characteristics sets* [58] and other *summarization techniques* [21, 59]. Additionally, more work is needed to investigate the impact of privacy [60] and security [61] during LTQP over decentralized environments. The incorporation of more expressive Linked Data Fragments interfaces [62, 63, 64, 65, 66, 67] in certain Solid vaults could introduce interesting trade-offs in terms of server and client query execution effort, especially if they can be combined in a heterogeneous manner [68, 69, 70].

In summary, traversal-based querying over decentralized environments can become practically feasible performance-wise. Furthermore, it is useful given the lack of alternatives, because centralization of private data may not be feasible or legal. However, for complex queries, more improvements through future research are needed.

This work provides answers to the increasing need of querying over decentralized environments, and uncovers next steps for resolving current limitations. Hence, it brings us closer to querying a decentralized Web where users are in full control.

Acknowledgements

This work is supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1274521N)

Supplemental Material Statement

Implementation: https://github.com/comunica/comunica-feature-link-traversal
Experiments: https://github.com/comunica/Experiments-Solid-Link-Traversal
Benchmark: https://github.com/SolidBench/SolidBench.js

References

1. Berners-Lee, T.J.: Information management: A proposal. (1989).
2. Verborgh, R.: Re-decentralizing the Web, for good this time. In: Seneviratne, O. and Hendler, J. (eds.) Linking the World's Information: A Collection of Essays on the Work of Sir Tim Berners-Lee. ACM (2022).
3. Bluesky: Bluesky. https://blueskyweb.xyz/ (2023).
4. Zignani, M., Gaito, S., Rossi, G.P.: Follow the Mastodon: Structure and Evolution of a Decentralized Online Social Network. In: Twelfth International AAAI Conference on Web and Social Media (2018).
5. Kuhn, T., Taelman, R., Emonet, V., Antonatos, H., Soiland-Reyes, S., Dumontier, M.: Semantic micro-contributions with decentralized nanopublication services. PeerJ Computer Science. (2021). doi:10.7717/peerj-cs.387
6. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G.de, Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., others: Knowledge graphs. Synthesis Lectures on Data, Semantics, and Knowledge. 12, 1–257 (2021).
7. Dedecker, R., Slabbinck, W., Wright, J., Hochstenbach, P., Colpaert, P., Verborgh, R.: What's in a Pod? – A knowledge graph interpretation for the Solid ecosystem. In: Saleem, M. and Ngonga Ngomo, A.-C. (eds.) Proceedings of the 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs. pp. 81–96 (2022).
8. Berners-Lee, T.: Linked Data. https://www.w3.org/DesignIssues/LinkedData.html (2009).
9. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1: Concepts and Abstract Syntax. W3C, https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (2014).
10. Feigenbaum, L., Todd Williams, G., Grant Clark, K., Torres, E.: SPARQL 1.1 Protocol. W3C, https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/ (2013).
11. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization Techniques for Federated Query Processing on Linked Data. In: International semantic web conference. pp. 601–616. Springer (2011).
12. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics. 37, 184–206 (2016).
13. Saleem, M., Ngomo, A.-C.N.: Hibiscus: Hypergraph-based Source Selection for SPARQL Endpoint Federation. In: European semantic web conference. pp. 176–191. Springer (2014).
14. Görlitz, O., Staab, S.: Splendid: SPARQL Endpoint Federation Exploiting Void Descriptions. In: Proceedings of the Second International Conference on Consuming Linked Data-Volume 782. pp. 13–24. CEUR-WS. org (2011).
15. Hartig, O.: An Overview on Execution Strategies for Linked Data Queries. Datenbank-Spektrum. 13, 89–99 (2013).

16. Hartig, O., Freytag, J.-C.: Foundations of Traversal based Query Execution over Linked Data. In: Proceedings of the 23rd ACM conference on Hypertext and social media. pp. 43–52. ACM (2012).
17. Speicher, S., Arwe, J., Malhotra, A.: Linked Data Platform 1.0. W3C, https://www.w3.org/TR/ldp/ (2015).
18. Turdean, T.: Type Indexes. Solid, https://solid.github.io/type-indexes/ (2022).
19. Hartig, O.: SPARQL for a Web of Linked Data: Semantics and computability. In: Extended Semantic Web Conference. pp. 8–23. Springer (2012).
20. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.-U., Umbrich, J.: Data summaries for on-demand queries over linked data. In: Proceedings of the 19th international conference on World wide web. pp. 411–420 (2010).
21. Umbrich, J., Hose, K., Karnstedt, M., Harth, A., Polleres, A.: Comparing data summaries for processing live queries over linked data. World Wide Web. 14, 495–544 (2011).
22. Hartig, O., Hose, K., Sequeda, J.: Linked Data Management. In: Sakr, S. and Zomaya, A. (eds.) Encyclopedia of Big Data Technologies. Springer, Germany (2019). doi:10.1007/978-3-319-63962-8_76-1
23. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: Extended Semantic Web Conference. pp. 154–169. Springer (2011).
24. Mendelzon, A.O., Mihaila, G.A., Milo, T.: Querying the world wide web. In: Fourth International Conference on Parallel and Distributed Information Systems. pp. 80–91. IEEE (1996).
25. Konopnicki, D., Shmueli, O.: Information gathering in the World-Wide Web: the W3QL query language and the W3QS system. ACM Transactions on Database Systems (TODS). 23, 369–410 (1998).
26. Chakrabarti, S., Van den Berg, M., Dom, B.: Focused crawling: a new approach to topic-specific Web resource discovery. Computer networks. 31, 1623–1640 (1999).
27. Batsakis, S., Petrakis, E.G.M., Milios, E.: Improving the performance of focused web crawlers. Data & Knowledge Engineering. 68, 1001–1013 (2009).
28. Hartig, O., Pirrò, G.: SPARQL with Property Paths on the Web. Semantic Web. 8, 773–795 (2017).
29. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C, https://www.w3.org/TR/2013/REC-sparql11-query-20130321/ (2013).
30. Bogaerts, B., Ketsman, B., Zeboudj, Y., Aamer, H., Taelman, R., Verborgh, R.: Link Traversal with Distributed Subweb Specifications. In: Rules and Reasoning: 5th International Joint Conference, RuleML+RR 2021, Leuven, Belgium, September 8 – September 15, 2021, Proceedings (2021).
31. Hartig, O., Özsu, M.T.: Walking without a Map: Optimizing Response Times of Traversal-based Linked Data Queries (extended version). arXiv preprint arXiv:1607.01046. (2016).
32. Schaffert, S., Bauer, C., Kurz, T., Dorschel, F., Glachs, D., Fernandez, M.: The linked media framework: Integrating and interlinking enterprise media content

and data. In: Proceedings of the 8th International Conference on Semantic Systems. pp. 25–32 (2012).
33. Hartig, O., Pérez, J.: LDQL: A query language for the web of linked data. Journal of Web Semantics. 41, 9–29 (2016).
34. Fionda, V., Pirrò, G., Gutierrez, C.: NautiLOD: A formal language for the web of data graph. ACM Transactions on the Web (TWEB). 9, 1–43 (2015).
35. Capadisli, S., Berners-Lee, T., Verborgh, R., Kjernsmo, K.: Solid Protocol. Solid, https://solidproject.org/TR/protocol (2020).
36. Van Herwegen, J., Verborgh, R., Taelman, R., Bosquet, M.: Community Solid Server. https://github.com/CommunitySolidServer/CommunitySolidServer (2022).
37. Inrupt: PodSpaces. https://docs.inrupt.com/pod-spaces/ (2022).
38. Flanders, D.: The Flemish Data Utility Company. https://www.vlaanderen.be/digitaal-vlaanderen/het-vlaams-datanutsbedrijf/the-flemish-data-utility-company (2022).
39. Capadisli, S.: Web Access Control. Solid, https://solid.github.io/web-access-control-spec/ (2022).
40. Bosquet, M.: Access Control Policy (ACP). Solid, https://solid.github.io/authorization-panel/acp-specification/ (2022).
41. Coburn, A., Pavlik, elf, Zagidulin, D.: Solid-OIDC. Solid, https://solid.github.io/solid-oidc/ (2022).
42. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems (TODS). 34, 1–45 (2009).
43. Hartig, O.: SQUIN: a traversal based query execution system for the web of linked data. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 1081–1084 (2013).
44. Ladwig, G., Tran, T.: SIHJoin: Querying remote and local linked data. In: Extended Semantic Web Conference. pp. 139–153. Springer (2011).
45. Miranker, D.P., Depena, R.K., Jung, H., Sequeda, J.F., Reyna, C.: Diamond: A SPARQL query engine, for linked data based on the Rete match. In: Proc. of the Workshop on Artificial Intelligence meets the Web of Data (AImWD) (2012).
46. Wilschut, A.N., Apers, P.M.G.: Pipelining in query execution. In: Proceedings. PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications. p. 562. IEEE (1990).
47. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a Modular SPARQL Query Engine for the Web. In: Proceedings of the 17th International Semantic Web Conference (2018).
48. Fafalios, P., Yannakis, T., Tzitzikas, Y.: Querying the Web of Data with SPARQL-LD. In: Research and Advanced Technology for Digital Libraries: 20th International Conference on Theory and Practice of Digital Libraries, TPDL 2016, Hannover, Germany, September 5–9, 2016, Proceedings 20. pp. 175–187. Springer (2016).
49. Hartig, O.: How caching improves efficiency and result completeness for querying linked data. In: LDOW (2011).

50. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.-D., Boncz, P.: The LDBC social network benchmark: Interactive workload. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 619–630 (2015).
51. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proceedings of the 13th International Conference on Database Theory. pp. 4–33. ACM (2010).
52. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimization using Selectivity Estimation. In: Proceedings of the 17th international conference on World Wide Web. pp. 595–604. ACM (2008).
53. Nielsen, J.: Response times: the three important limits. Usability Engineering. (1993).
54. Deshpande, A., Ives, Z., Raman, V.: Adaptive query processing. Foundations and Trends\textregistered in Databases. 1, 1–140 (2007).
55. Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: an adaptive query processing engine for SPARQL endpoints. In: International Semantic Web Conference. pp. 18–34. Springer (2011).
56. Acosta, M., Vidal, M.-E.: Networks of linked data eddies: An adaptive web query processing engine for RDF data. In: International Semantic Web Conference. pp. 111–127. Springer (2015).
57. Heling, L., Acosta, M.: Robust query processing for linked data fragments. Semantic Web. 1–35 (2022).
58. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: 2011 IEEE 27th International Conference on Data Engineering. pp. 984–994. IEEE (2011).
59. Prud'hommeaux, E., Bingham, J.: Shape Trees Specification. W3C, https://shape-trees.org/TR/specification/ (2021).
60. Taelman, R., Steyskal, S., Kirrane, S.: Towards Querying in Decentralized Environments with Privacy-Preserving Aggregation. In: Proceedings of the 4th Workshop on Storing, Querying, and Benchmarking the Web of Data (2020).
61. Taelman, R., Verborgh, R.: A Prospective Analysis of Security Vulnerabilities within Link Traversal-Based Query Processing. In: Proceedings of the 6th International Workshop on Storing, Querying and Benchmarking Knowledge Graphs (2022).
62. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a low-cost Knowledge Graph Interface for the Web. Journal of Web Semantics. 37, 184–206 (2016).
63. Azzam, A., Fernández, J.D., Acosta, M., Beno, M., Polleres, A.: SMART-KG: hybrid shipping for SPARQL querying on the web. In: Proceedings of The Web Conference 2020. pp. 984–994 (2020).
64. Minier, T., Skaf-Molli, H., Molli, P.: SaGe: Web preemption for public SPARQL query services. In: The World Wide Web Conference. pp. 1268–1278 (2019).

65. Azzam, A., Aebeloe, C., Montoya, G., Keles, I., Polleres, A., Hose, K.: WiseKG: Balanced access to web knowledge graphs. In: Proceedings of the Web Conference 2021. pp. 1422–1434 (2021).
66. Aebeloe, C., Keles, I., Montoya, G., Hose, K.: Star Pattern Fragments: Accessing Knowledge Graphs through Star Patterns. arXiv preprint arXiv:2002.09172. (2020).
67. Hartig, O., Buil-Aranda, C.: Bindings-restricted triple pattern fragments. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 762–779. Springer (2016).
68. Heling, L., Acosta, M.: Federated SPARQL Query Processing over Heterogeneous Linked Data Fragments. In: Proceedings of the ACM Web Conference 2022. pp. 1047–1057 (2022).
69. Cheng, S., Hartig, O.: FedQPL: A Language for Logical Query Plans over Heterogeneous Federations of RDF Data Sources. In: Proceedings of the 22nd International Conference on Information Integration and Web-based Applications & Services. pp. 436–445 (2020).
70. Montoya, G., Aebeloe, C., Hose, K.: Towards efficient query processing over heterogeneous RDF interfaces. In: 2nd Workshop on Decentralizing the Semantic Web, DeSemWeb 2018. CEUR Workshop Proceedings (2018).