FACULTY OF ENGINEERING
AND ARCHITECTURE

Abstracting Data Updates over a Document-oriented interface of a Permissioned Decentralized Environment

Jitse De Smet
Student number: 01901393

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh
Counsellors: Dr. ir. Ruben Taelman, Bryan-Elliott Tam

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2023-2024

GHENT
UNIVERSITY

Foreword

I would like to acknowledge everyone who accompanied me during my Bachelor and Master. My dedication and their help during my master's and bachelor's degrees allowed me to expand my knowledge. Allowing me to grow both individually and academically. Reducing the scope to this Master dissertation, I would specifically like to thank my promotors Dr. ir. Ruben Taelman and Prof. dr. ir. Ruben Verborgh for expecting the best in me and supporting me academically. Beyond their academic support, they have gone beyond what was expected of them and also allowed me to talk to them about life in general. In my personal spheres, I would like to explicitly thank my parents and my girlfriend, as well as my friends.

Abstracting Data Updates over a Document-oriented interface of a Permissioned Decentralized Environment

Jitse De Smet
jitse.desmet@ugent.be
Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering
Academic Year 2023-2024
Faculty of Engineering and Architecture
Ghent University

Supervisors:	Counsellors:
Dr. ir. Ruben Taelman	Dr. ir. Ruben Taelman
Prof. dr. ir. Ruben Verborgh	Bryan-Elliott Tam

Keywords - Semantic Web, Update Queries, Solid

Data is the new gold; you hear it constantly. Much of that gold flows through Web technologies into the centralized data stores of massive companies such as Amazon, Google, and TikTok. The Web, however, was envisioned as a decentralized information space to which anyone could read and write information. Today's centralization of data causes numerous problems, such as privacy-related issues and the centralization of attention. This centralization of attention and media control causes social turbulence. For example, the US ban on TikTok or, more recently, the ban of TikTok by France in response to protests. In response to these crises, various initiatives, such as Solid and Mastodon, are working towards re-decentralizing the Web. The re-decentralization of the Web comes with various challenges to overcome, since the world is not the same as it used to be. These challenges range from efficient and interoperable reading and writing to expressing potentially complex usage/ access policies. Efficient reading in the context of a decentralized, permissioned ecosystem has received some research attention, but writing remains rather unexplored. Therefore, we examined the current state of the Solid specification to investigate the problems and data dependencies updates currently face. The most challenging problem is access path dependence, where writers of data need to explicitly specify a location to write or update data. Similar issues are present when reading data in Solid, but they are abstracted through the use of a query engine. We therefore investigate the possibility of a query engine that can create and update resources without data dependencies. Our evaluations show that such a query engine can be created by providing a structural description and has limited overhead. Having a data dependency free approach to update decentralized data is of massive importance in the adaptation of decentralized systems, as it allows easier management of data. The current implementation is limited to updating data of one federation, additional research is required to support inter-federation updates.

Abstracting Data Updates over a Document-oriented interface of a Permissioned Decentralized Environment

Jitse De Smet

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. ir. Ruben Verborgh

Abstract - **Data is the new gold; you hear it constantly. Much of that gold flows through Web technologies into the centralized data stores of massive companies such as Amazon, Google, and TikTok. The Web, however, was envisioned as a decentralized information space to which anyone could read and write information. Today's centralization of data causes numerous problems, such as privacy-related issues and the centralization of attention. This centralization of attention and media control causes social turbulence. For example, the US ban on TikTok or, more recently, the ban of TikTok by France in response to protests. In response to these crises, various initiatives, such as Solid and Mastodon, are working towards re-decentralizing the Web. The re-decentralization of the Web comes with various challenges to overcome, since the world is not the same as it used to be. These challenges range from efficient and interoperable reading and writing to expressing potentially complex usage/ access policies. Efficient reading in the context of a decentralized, permissioned ecosystem has received some research attention, but writing remains rather unexplored. Therefore, we examined the current state of the Solid specification to investigate the problems and data dependencies updates currently face. The most challenging problem is access path dependence, where writers of data need to explicitly specify a location to write or update data. Similar issues are present when reading data in Solid, but they are abstracted through the use of a query engine. We therefore investigate the possibility of a query engine that can create and update resources without data dependencies. Our evaluations show that such a query engine can be created by providing a structural description and has limited overhead. Having a data dependency free approach to update decentralized data is of massive importance in the adaptation of decentralized systems, as it allows easier management of data. The current implementation is limited to updating data of one federation, additional research is required to support inter-federation updates.**

Keywords - Semantic Web, Update Queries, Solid

I. Introduction

Data in today's web is increasingly captured in huge data silos. The extent of these silos is enormous, reaching the limits of what is societal and legislative permitted. From a societal perspective, these silos are a giant threat to the privacy of users. This centralization of privacy causes social turbulence, since it centralizes the attention of the masses and thus media control into a select few. Luckily, legislative measures have been taken to protect society from this centralization [1] [2] . As a response, centralization technologies are being developed, such as Solid [3] , Bluesky [4] , Mastodon [5] and various blockchain-based initiatives [6] .

The Solid initiative achieves decentralization by creating a standard building on top of existing Web standards. This approach allows for interoperability and easier workflow adaptation by leveraging existing expertise. Nevertheless, the re-decentralization of the Web comes with various challenges ranging from efficient and effective read and write operations, to expressing and enforcing access and usage control policies. Reading data in this context has already gained some scientific attention [7] [8] , but effectively writing data remains rather unexplored.

Data decentralization initiatives such as Solid and Bluesky decentralize data by providing each user with a self-governed data store. Users are in control of their data store, how they interact with the datastore and who they share their data with. The effectiveness of reading data in a decentralized environment has been increased by abstracting data reads through a query abstraction layer, the query engine, by using query languages like GraphQL [9] and SPARQL [10] . In this work, we will similarly research how we can abstract data updates by using a query abstraction layer. The current (draft) Solid specification [11] describes each data store, or pod, as a document oriented interface where a user decides for each document who can access that document. Our goal is thus to create a query engine that effectively decides what document a resource should be stored in, liminating the access-path data dependency. We hypothesize that such a query engine has a 2x overhead in the number of HTTP requests and a 4x overhead in the execution time compared to a query engine that requires the user to configure the document explicitly. This overhead is often acceptable because applications are typically created in such a way that they synchronize local changes in the background, without disturbing the user. This acceptable delay of updates contrasts with reads because in the case of reading data, the user flow is often interrupted when information is transferred.

II. Related Work

The Solid specification [11] builds on top of existing Semantic Web technologies such as RDF (Resource Description Framework) [12] and LDP (Linked Data Platform) [13] . LDP is a set

J. De Smet is a master student with the KNowledge On Web Scale team within IDLab, Ghent University (UGent), Gent, Belgium. Email: jitse.desmet@ugent.be

of rules that is used to create a document oriented interface acceptable through HTTP. Such an interface essentially exposes a file system over HTTP, it creates directories, called Containers, that group together data documents and directories. Each of the exposed HTTP resources has their own access control policy declared through either WAC [14] or ACP [15] .

A. Theoretical positioning of Solid

The collection of all Solid pods can be interpreted as one big permissioned decentralized graph database with some interesting properties. A typical distributed database will both replicate and shard its data [16] . Sharding data means that the collection of all data is divided into smaller shards, and each machine manages one or more shards. Sharding allows us to scale our data horizontally. Each shard is then replicated on multiple machines, making the system partition tolerant [17] . Different approaches exist to configure the shards and replications. Often times, each shard will have one leader replication, and the other replications are followers. Reads then happen to both leader and followers, while writes only happen to the leader. The leader is responsible for synchronizing all changes to the followers. Such a configuration chooses reads to have eventual consistency [16] [18] , positioning itself on the CAP scale [17] [19] by choosing Availability and Partition Tolerance.

The Solid specification builds on top of HTTP and therefore, links can break [20] . This essentially means that there is no partition tolerance. When a pod is disconnected, the data on that pod becomes unavailable. Solid thus only has sharding and no replication from a theoretical perspective. This is an interesting design choice because it means that the Solid specification is yet to position itself on the CAP scale.

B. Concise Bounded Description

In this work, we will try to store RDF resources, defined as the CBD (Concise Bounded Description) [21] of a Named Node. The CBD of a resource is defined as the collection of triples that can be accessed by recursively following objects, without following named nodes. As an example, Figure 1 contains some RDF data in turtle [22] format, taking the CBD of named node `<me>` results in Figure 2.

C. Resource Descriptions

RDF datastores, and by extension, Solid pods, are schemaless. This means that data contained does not follow a specific, rigid format unlike a relational database. In the context of Big Data, this is beneficial because defining a schema that should be followed by all actors that write data is impossible, since both the actors, and their way of working constantly changes. Data consumers might expect data they consume to follow a certain format. Shape descriptions describe the format of data and can be used to validate that data indeed follows the expected format. Two RDF shape description languages are important, ShEx [23] and SHACL [24] . A self-descriptive declaration of ShEX can be found in Figure 3. We do not provide a similar example for SHACL since its syntax uses turtle. This is more verbose, but the predicate names are self-describing.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<me> a foaf:Person ;
  foaf:givenName "Alice" ;
  foaf:knows ex:Bob, ex:Carol ;
  foaf:knows [
      foaf:givenName "Dave"
    ] .
ex:Bob
  foaf:givenName "Bob" ;
  foaf:familyName "Builder" .
```

Figure 1: An example RDF file.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<me> a foaf:Person ;
  foaf:givenName "Alice" ;
  foaf:knows ex:Bob, ex:Carol ;
  foaf:knows [
      foaf:givenName "Dave"
    ] .
```

Figure 2: The CBD of named node <me> in Figure 1.

```
# our EmployeeShape reuses the FOAF ontology
# An <EmployeeShape> has:
<EmployeeShape> {
# at least one givenName.
  foaf:givenName xsd:string+,
  foaf:familyName xsd:string, # one familyName.
  foaf:phone IRI*, # any number of phone numbers.
  foaf:mbox IRI # one FOAF mbox.
}
```

Figure 3: Self-explanetory example ShEx shape.

D. Storage Organization Descriptions

Just like RDF does not define its data schema, so does LDP not define its data organization. As a result, someone reading a pod does not know where it can find the data relevant to them. However, just like RDF data can be described using a resource description, so can an LDP interface organization be described. Solid proposes two ways of describing a pod: Type Indexes [25] and Shape Trees [26] . The Type Indexes specification was a first attempt at describing the resources of a Pod. It describes the use of a public and private index over the `rdf:type` predicate. The construction of a public and private index is, however, fundamentally flawed since resources cannot be grouped into either public or private because more complex access control is the norm.

Shape Trees are the proposed replacement of Type Indexes. Type Indexes were limited to creating indexes based on the type predicate. As an improvement Shape Trees index is based on some Resource Description. Moreover, Shape Trees are the natural extension of these resource descriptions to resource hierarchies.

In the context of read queries, it has been proven that using the structure of a pod by, for example, consulting the type indexes can be beneficial [8] . It thus makes sense to speculate that a similar structural description can help write queries.

III. Storage Guidance Vocabulary

Unfortunately, neither Type Indexes [25] , nor Shape Trees [26] are sufficiently descriptive to assess whether a resource should be stored in a document. As an example, we give a small list of questions that cannot be answered by either data store descriptions:

1. What if multiple directories match? Do I duplicate the resource?
2. What should I do if no documents match?
3. How are resources grouped?
   i. Can I infer that resources grouped by a property are always grouped by that property?
   ii. Does that mean that if I get a new object for that property that I can just create a new document?
4. What should I do when I update a resource?
   i. Should I alter the data store description?
   ii. Should I move the resource? (Assign a new named node?)
5. Are all clients equal? Do they all abide the structural information description?

To answer these questions, we developped a new vocabulary, namely, the Storage Guidance Vocabulary (SGV). The basic concepts of the vocabulary are:

Resource Collection: Corresponds to a group of RDF resources.

Unstructured Collection: Corresponds to a classical LDP container or HTTP resource.

Structured Collection: A canonical or derived collection. (below)

Canonical Collection: A resource collection containing resources.

Derived Collection: A resource collection that stores resources already stored by one or more other structured containers.

Resource Description: A way of describing resources, for example through ShEx or SHACL.

Group Strategy: A description of how resources should be grouped together, for example: my images are grouped per creation date.

Store Condition: When multiple collections are eligible to store a resource, the store condition decides what collection(s) actually store the resource. Allowing the creation of a store priority system.

Update Condition: Describes what to do when a containing resource is changed.

Client Control: Describes the amount of freedom a client has when trying to store a resource.

A. Flow: Create Resource

To clarify the different concepts, we will walk through an example flow to create a resource. Figure 4 shows a query that would trigger this flow. The query engine will essentially discover what URI should be used as a base. To do so, it goes through the followings steps:

1. The client gets the SGV description of the storage space (can be cached).

```
INSERT DATA {
  <> a ns1:Post ;
    ns1:content
      "I want to eat an apple." ;
    ns1:creationDate
"2024-05-08T23:23:56Z"^^xsd:dateTime ;
    ns1:id "416608218494388"^^xsd:long ;
    ns1:hasCreator card:me ;
    ns1:hasTag tag:Austria ;
    ns1:isLocatedIn resource:China .
}
```

Figure 4: Example resource insertion query

2. The client checks all canonical collections and checks if the resource to be inserted matches a resource description of the collection.
3. If the resource matches a description, the client checks the store condition of the description given the eligible collections.
4. For each collection that stores the resource:
   i. The client checks the group strategy of the collection and groups the resource accordingly, deciding on the name of the new resource.
   ii. The client checks the collections that are derived from this collection. Step 4 is executed for all collections that are derived from this collection, and the resource matches the description.
5. The client performs the store operation.

B. Flow: Update Resource

Creating a resource is, of course, only one facet. Now we will look at the flow of updating, or similarly, deleting data. Figure 5 shows an example update query that would trigger an update flow:

1. The client gets the SGV description of the storage space and the HTTP resource containing the updated RDF resource.
2. The client virtually constructs the resource that would result from the requested operation.
3. The client checks the update condition of the original matching resource description. The following action depends on the update condition. Typically, the update-condition will say whether an RDF resource is moved or not.
   i. Move required: remove the existing resource and follow the steps described in the create resource flow.
   ii. No move required: just update the resource as requested by the user.

```
DELETE {
  ?id ns1:id "416608218494388"^^xsd:long .
} INSERT {
  ?id ns1:id "416608218494389"^^xsd:long .
} where {
    BIND(:416608218494388 as ?id)
}
```

Figure 5: Example resource update query

IV. Evaluation

To verify our hypothesis, we implemented an SGV aware query engine and benchmarked it[1]. The provided implementation does not implement all features of SGV, but provides a sufficient implementation to verify the hypothesis. Our hypothesis is two-fold. We want to verify both the HTTP-request overhead and the execution time overhead. The former will be evaluated theoretically as it provides more insights into the system. The latter, on the other hand, will require an empirical evaluation.

A. Theoretical Evaluation

Depending on the operation, a different number of HTTP requests is required. We will analyse three different cases: 1. creating a resource, 2. updating a resource, but not moving it, and 3. updating a resource and moving it.

1. Creating a Resource

The creation of a resource requires the client to fetch the SGV description. Assuming the description is located in a single file, this amounts to one HTTP request.

After getting the description, the client computes the location the resource should be stored. When done, the client performs another HTTP request to store the resource there.

Our hypothesis holds for this operation, since a client not using SGV would require but one HTTP request, namely to create the resource.

2. Updating a resource, not moving it

In the case of updating a resource, we need to both get the SGV description and the original resource. Getting both resources can, however, be done in parallel.

A client will now compute whether the resource should be moved, and conclude it need not be moved. The client will thus perform another request to update the HTTP resource it just received.

In total, this amounts to tree HTTP requests. A non-SGV aware client would require two HTTP requests, one to get the original resource in order to create its binding, and one to perform the update. The hypothesis thus holds true.

3. Updating a resource, moving it

In case a move is required, an SGV engine would do the same steps as before, but when updating the resource, it would need two (parallel) HTTP requests. One to delete the original and one to create the new one. Resulting in four HTTP requests. A non-SGV query engine would require at least three HTTP requests to achieve the same result, two of which can also be parallelized. This thus verifies our HTTP requests count hypothesis.

B. Empirical evaluation

For our empirical evaluation, we benchmark our SGV-aware engine using SolidBench[2]. This exposes pods containing LDBC Social Network Benchmark (SNB) [27] data. We created four pods with their own way of organizing social media posts: 1. organizing all posts in a directory with a file for each creation date, 2. organizing all posts in a directory with a file for each creation location, 3. organizing all posts in a directory with a file post, and 4. organizing all posts in one file.

These organization structures are then evaluated using queries that test five different choke points: 1. creating a resource, 2. updating a resource, not modifying it, 3. updating a resource, moving it, 4. performing an illegal update, and 5. deleting a resource.

Choke points 1 and 3 are most essential for SGV, we therefore provide the measure execution time for these in respectively Table 1 and Table 2. The interested reader can find the other evaluations in the accompanying document.

From evaluations, we concluded that our hypothesis holds when we compare the execution time of an SGV query engine to a non-SGV engine that executes the same operations. Note that using a non-SGV engine, one cannot express the resource move using a single SPARQL query because SPARQL cannot express the CBD.

Table 1: Average execution time of inserting data (avg. of 100 runs)

frag. Strat.	ops/sec	Average Time (ms)	Margin
by date: SGV	22	44582.068	±1.73%
by date: RAW	35	27899.513	±2.07%
one file: SGV	6	149415.739	±2.98%
one file: RAW	7	134361.192	±8.66%
own file: SGV	10	91851.395	±2.56%
own file: RAW	13	76672.217	±3.07%
by location: SGV	23	43005.366	±2.20%
by location: RAW	35	28003.949	±2.53%

V. Future Work

To our knowledge, this is the first effort of abstracting data updates over a document-oriented interface of a decentralized permissioned environment. As such, there is a plenty of future work.

Table 2: Average execution time of moving a resource (avg. of 100 runs)

Task	ops/sec	Average Time (ms)	Margin
by date: SGV	7	141940.530	±1.28%
by date: RAW	11	87113.119	±0.75%
one file: SGV	2	343690.220	±1.70%
one file: RAW	4	208930.211	±2.04%
own file: SGV	5	177991.908	±0.58%
own file: RAW	12	80729.940	±1.06%
by location: SGV	7	133052.120	±0.60%
by location: RAW	12	81066.196	±1.15%

[1]Both the implementation, and the benchmark are available at: https://github.com/jitsedesmet/sgv-update-engine/releases/tag/v0.0.2

[2]https://github.com/SolidBench/SolidBench.js

A. Inter-Pod updates

In this work we reduced the complexity by assuming that we want to update a single pod. Of course, updating multiple pods with a single query is a logical next step where different considerations need to be taken into account.

1. As a pod owner, I want to transfer the pictures I have to someone else, so they now own that picture. Note that I am not guaranteed to have write permissions to the other Solid pod.
2. As a pod owner, I want to transfer a token to a pod I do, or do not, have write access to. The token should always exist *exactly once*, meaning there is always one person holding the token, and everyone can see who has it.
3. As a pod owner, I want to insert an additional property to an existing resource in someone else's pod. For example, I transferred a picture and forgot to add a description.
4. As a pod owner, I want to delete a property of an existing resource in someone else's pod.
5. As a pod owner, I want to remove a resource in someone else's pod, so I don't see it any more. Essentially, I want to change my view over the resource. This could be achieved by using the Subweb Specification [28] and adding a rule that makes me ignore the "virtually" deleted triple.
6. As a pod owner, I want to remove a resource in someone else's pod, so no one can see it. I might want to send a suggestion in a notification collection of the targetted pod.

B. Other Interfaces

In this work, we investigate document-oriented interfaces, focussing on LDP. With document-oriented interface, the question when inserting a resource is mostly: "Where do I store this resource?" When using a different interface, that question might shift to "What other resources are linked to this new one, and though what links?" There is merit in investigating different interfaces for the use of decentralized data storages, as document oriented interfaces come with drawbacks [29] . Beyond the drawbacks listed there, much of the complexities of SGV are a result of the unordered, document oriented nature of SGV. This non-descriptiveness, however, is at the benefit of the data provider, and thus it's unlikely that LDP will disappear.

Another interesting approach would be to create multiple interfaces on the same data, as an example one interface would serve as a SPARQL endpoint. Another endpoint could be an LDP interface that derives collections based on the canonical collection that is the SPARQL endpoint.

C. View Creation And Discovery

Derived resources have already proved to be beneficial to solve issues of LDP [30] . In our benchmarks, we see that the pod data organization heavily influences the execution time of queries. The application exposing the data could infer what kind of resource organization would benefit clients through usage statistics. The server could then decide to create a derived collection to enable faster query execution.

D. Smart Access Control

Access control policies are currently created per document. This makes access control difficult to understand, since it is not clear why a single document has a certain access policy. SGV describes why and what data is stored in a certain document. Configuring an access policy in a certain document can thus be translated to what kind of resources follow what policy. Extracting policies based on the data can be useful when derived resources come into play. For example, it could be inferred when you have access to some resource in a canonical collection, that you should also have access to that resource in a derived collection given no data enrichment happened when the derived resource was created.

VI. Conclusion

In this work, we presented a vocabulary that allows smart clients to autonomously discover the location a created or updated resource should be stored within a document oriented decentralized interface of a permissioned decentralized environment. The vocabulary also introduces checks on whether a resource can be created or removed. Additionally, we proved that our vocabulary is indeed expressive by implementing a smart client that consumes it.

We hypothesized that such a smart client would be a maximum of four times slower and would require a maximum of double the amount of HTTP requests. Through theoretical evaluation, we discovered that the amount of HTTP requests is within those bounds. Using empirical evaluation, we also validated that the execution time overhead is within the accepted range. Moreover, we saw that some of SGVs behaviour cannot be modelled using a SPARQL query.

In essence, SGV tries to provide structure to a widely unstructured document store, namely LDP. It does this by defining a server-side description of the structure that should be enforced by clients. In reality, clients can still interact with the data store however they want, since the server is not aware that a structure should be followed. This lack of server-side verification is perhaps the biggest shortcoming of this work. That being said, it is entirely possible to extend an existing data store server with an SGV verification system. The downside at that being that both the client and server need to calculate the proposed location of a resource. Unfortunately, this is a shortcoming of choosing for a low-complexity server, only to later conclude you want to assert complex guarantees. Additionally, since one server interface is used by many decentralized clients, it becomes almost impossible to guarantee a system that respects the structure of a permissive interface without creating server-side validation.

Acknowledgements

I would like to acknowledge everyone who accompanied me during my Bachelor and Master. My dedication and their help during my master's and bachelor's degrees allowed me to expand my knowledge. Allowing me to grow both individually and academically. Reducing the scope to this Master dissertation, I would specifically like to thank my promotors Dr. ir. Ruben Taelman and Prof. dr. ir. Ruben Verborgh for expecting the best in me and supporting me academically. Beyond their academic support, they have gone beyond what was expected of them and also allowed me to talk to them about life in gen-

eral. In my personal spheres, I would like to explicitly thank my parents and my girlfriend, as well as my friends.

REFERENCES

[1] European Parliament and Council of the European Union, “Regulation (EU) 2016/679 of the European Parliament and of the Council.” [Online]. Available: https://data.europa.eu/eli/reg/2016/679/oj

[2] California State Legislature, “The California Consumer Privacy Act of 2018.” [Online]. Available: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375

[3] E. Mansour et al., “A Demonstration of the Solid platform for Social Web Applications,” in Proceedings of the 25thInternational Conference Companion on World Wide Web,2016, pp. 223–226.

[4] M. Kleppmann et al., “Bluesky and the AT Protocol: Usable Decentralized Social Media.” 2024.

[5] M. Zignani, S. Gaito, and G. P. Rossi, “Follow the “Mastodon”: Structure and Evolution of a Decentralized Online Social Network,” in Twelfth International AAAI Conference on Web and Social Media, 2018.

[6] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Decentralized business review, 2008.

[7] O. Hartig and M. T. Özsu, “Walking without a map: Ranking-based traversal for querying linked data,” in The Semantic Web–ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I 15, 2016, pp. 305–324.

[8] R. Taelman and R. Verborgh, “Link traversal query processing over decentralized environments with structural assumptions,” in International Semantic Web Conference, 2023, pp. 3–22.

[9] Facebook and GraphQL contributors, “GraphQL spec October 2021 Edition,” Oct. 2021.

[10] A. Seaborne and S. Harris, “SPARQL 1.1 Query Language,” Mar. 2013.

[11] S. Capadisli, T. Berners-Lee, R. Verborgh, and K. Kjernsmo, “Solid Protocol,” Dec. 2022.

[12] D. Wood, M. Lanthaler, and R. Cyganiak, “RDF 1.1 Concepts and Abstract Syntax,” Feb. 2014.

[13] S. Speicher, J. Arwe, and A. Malhotra, “Linked Data Platform 1.0,” Feb. 2015.

[14] T. Berners-Lee, H. Story, and S. Capadisli, “Web Access Control,” May 2024.

[15] M. Bosquet, “Access Control Policy (ACP),” May 2022.

[16] R. Elmasri, “Fundamentals of database systems seventh edition,” 2021.

[17] A. Fox and E. A. Brewer, “Harvest, yield, and scalable tolerant systems,” in Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, 1999, pp. 174–178.

[18] D. Pritchett, “BASE: An ACID Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability.,” Queue, vol. 6, no. 3, pp. 48–55, 2008.

[19] E. Brewer, “CAP twelve years later: How the “rules” have changed,” Computer, vol. 45, no. 2, pp. 23–29, Feb. 2012, doi: 10.1109/MC.2012.37.

[20] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann, “World-Wide Web: the information universe,” Internet Research, vol. 2, no. 1, pp. 52–58, 1992.

[21] P. Stickler, “CBD - Concise Bounded Description,” Jun. 2005.

[22] G. Carothers and E. Prud’hommeaux, “RDF 1.1 Turtle,” Feb. 2014.

[23] T. Baker and E. Prud’hommeaux, “Shape Expressions (ShEx) 2.1 Primer,” Oct. 2019.

[24] H. Knublauch and D. Kontokostas, “Shapes Constraint Language (SHACL),” Jul. 201

[25] T. Turdean, J. Zucker, V. Balseiro, S. Capadisli, and T. Berners-Lee, “Type Indexes,” Jun. 2022.

[26] E. Prud’hommeaux and J. Bingham, “Shape Trees Specification,” Dec. 2021

[27] O. Erling et al., “The LDBC social network benchmark: Interactive workload,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 619–630.

[28] B. Bogaerts, B. Ketsman, Y. Zeboudj, H. Aamer, R. Taelman, and R. Verborgh, “Link Traversal with Distributed Subweb Specifications,” in Proceedings of the 5th International Joint Conference on Rules and Reasoning, S. Moschoyiannis, R. Peñaloza, J. Vanthienen, A. Soylu, and D. Roman, Eds., in Lecture Notes in Computer Science, vol. 12851. Springer, Sep. 2021, pp. 62–79. doi: 10.1007/978-3-030-91167-6_5.

[29] R. Dedecker, W. Slabbinck, J. Wright, P. Hochstenbach, P. Colpaert, and R. Verborgh, “What’s in a Pod? A knowledge graph interpretation for the Solid ecosystem,” in 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs (QuWeDa) at ISWC 2022, 2022, pp. 81–96.

[30] J. Van Herwegen and R. Verborgh, “Granular Access to Policy-Governed Linked Data via Partial Server-Side Query.”

Contents

List of Figures

List of Tables

List of listings

List of Acronyms

***ACID* – atomicity, consistency, isolation, durability**: [1] 59.

***ACL* – Access Control List**. 15.

***ACP* – Access Control Policy**: [2] 16., 58.

***API* – Application Programming Interface**. 41.

***CAP* – Consistency, Availability, Partition tolerance**: [3] 14., 15., 59., 60.

***CBD* – Concise Bounded Description**: [4] 7., 8., 32., 51., 53., 54., 61.

***CCPA* – California Consumer Privacy Act**. i

***CRDT* – Conflict-free Replicated Data Type**: [5] 59., 60.

***GDPR* – General Data Protection Regulation**. i

***HTTP* – Hypertext Transfer Protocol**. ii, iii, iv, 6., 12., 17., 18., 22., 24., 27., 30., 39., 40., 41., 42., 43., 44., 46., 51., 53., 55., 60., 61.

***LDES* – Linked Data Event Streams**: [6] ii, 39.

***LDP* – Linked Data Platform**: [7] ii, iii, 12., 14., 17., 18., 22., 27., 30., 41., 56., 57., 58., 61.

***RDF* – Resource Description Framework**: [8] 6., 7., 8., 10., 11., 12., 14., 17., 18., 22., 24., 27., 29., 30., 31., 32., 33., 34., 36., 43., 44., 45., 58., 61.

***SGV* – Storage Guidance Vocabulary**: The vocabulary introduced in this document in order to explicetly dezscribe the structure of a Solid pod. 22., 23., 24., 30., 32., 37., 38., 39., 40., 41., 43., 44., 45., 48., 49., 51., 52., 53., 55., 56., 57., 58., 61.

SHACL: SHACL [9] is a W3C Recomendation shape description language. 11., 18., 23., 31., 35., 36., 40.

ShEx: ShEx [10] is a communty created shape description language. 11., 18., 23., 26., 31., 40., 45.

***SNB* – Social Network Benchmark**: [11] 20.

SPARQL. iii, 8., 9., 10., 11., 12., 32., 33., 34., 37., 42., 47., 51., 53., 54., 57., 61.

***TPF* – Triple Patter Fragments**: [12] 11.

TREE: [13] ii, 58.

URI. 6., 7., 31., 32., 40., 42., 48.

***VoID* – Vocabulary of Interlinked Datasets**: [14] ii

***W3C* – World Wide Web Consortium**. 6., 11.

***WAC* – Web Access Control**: [15] 15., 16., 58.

1 Preface

1.1 Introduction

The Web has become a primary driver for economic, scientific, and societal progress. This Web was envisioned as a globally interconnected decentralized information space against which anyone can read and write information. However, today's Web has become increasingly centralized, as most of the data is centralized in a few large data stores which are in full control of massive companies such as Amazon and Google. This centralization of data on the Web leads to numerous problems, such as privacy-related issues as **people are not in control of their own personal data** [16].

To solve these problems caused by data centralization, various initiatives are working towards *re-decentralizing* data on the Web, such as Solid [17], Bluesky [18], Mastodon [19], and various blockchain-based initiatives [20]. These initiatives allow people to choose where their data is stored, either in personal data vaults [17], shared federation instances [19], or publicly [20]. Blockchains, despite their popularity, are less suitable as a data management system because all records are public and shared among nodes. Furthermore, the computational cost is substantial. Recent privacy scandals and emerging legislation such as GDPR (*General Data Protection Regulation*) and CCPA (*California Consumer Privacy Act*) are leading to increasing adoption of these decentralization initiatives. Various companies and organizations world-wide are starting to build products and services on top of decentralization techniques, specifically Solid, such as *BBC* (UK), *Digita* (Flanders) and *Inrupt* (USA). The **fundamental shift** from centralized data management comes with various challenges. The Solid community tries to tackle these challenges through a specification. Interestingly, a rather unexplored domain is that of effectively writing data even though effective reads have achieved some attention. In this thesis, I will explore how we can **abstract data updates**, with a focus on Solid.

1.2 Problem Statement

Solid can theoretically be described as a permissioned decentralized data store. This large data store is split into many different pods that are individually governed. Contrary to widely used NoSQL databases, Solid does not create shards over these pods. Instead, when a pod experiences network partitioning, Solid accepts that the data on that pod is not reachable. This works because the underlying Linked Data principles always assume an open world principle, meaning that when data is not found, it doesn't conclude that it does not exist. Solid also deviates from blockchain because of this, nodes do not contain all data of the system, but only a fraction. Additionally, access control, and even usage control are of great importance to Solid.

Much research has been done as to how we can efficiently read from these pods. This research asks both how to answer a query as completely as possible over different pods, and also how to query a single pod efficiently. Solid currently describes only a single interface type, LDP (*Linked Data Platform*).

The idea of LDP is to map a simple, document oriented file structure to a Linked Data interface over HTTP (*Hypertext Transfer Protocol*). The interface allows for a simple server implementation and limited computational overload for servers. This means that much client-side logic is required to take up part of the search effort. To make matters worse for client-side searching, the interface structure depends on data provider preferences and does not need to be described by LDP. We require an abstraction layer to shield developers from these complexities. These abstractions can happen through query engines that fetch data requested through a declarative query.

These query engines already allow developers to query pods effectively through a technique called link traversal querying. A developer gives the root path of a Solid pod and the engine queries the whole pod. Even though it's effective, it could still benefit from efficiency improvements. Speed-ups can be gained by incorporation the structure/ organization of the pod in the query evaluation [21]. This organization can be described through different vocabularies, examples include, Type Index [22], Shape Trees [23], VoID (*Vocabulary of Interlinked Datasets*) [14], TREE [13], and LDES (*Linked Data Event Streams*) [6].

A pod can thus be organized, and since LDP maps to a file system, everyone, from data consumer, data producer and data owner, benefit from a good organization. Unfortunately, as of currently, there are no automated clients that infer where to store a resource in a way that does not break the organization. Developers that want to write data to a pod thus need to have numerous checks in place. Often times, they either break the organization or store their data in a hard-coded location and then alter the organization description to be compliant with the new organization. This way of working with data is unmaintainable, and it's precisely these data dependencies that caused Knuth to create the relational database [24].

In this thesis, we look at how we can create a query engine that can, in an application agnostic way, infer where a resource should be stored in a Solid pod. The scope of this thesis is limited to Solid and the LDP interface.

1.3 Research Question

The research question for this thesis is: **How can we abstract data updates over a document oriented interface of a permissioned decentralized environment behind a query abstraction layer?** We quickly go over the different terms in that question.

- Abstract data updates: We aim to abstract the query process, so a developer does not need to interact with the pod interfaces directly.
- Document oriented interface: the interface we interact with exposes data through HTTP documents.
- Permissioned: each HTTP resource has access rights configured, these rights can either grant or deny access to resources for specific users.
- Decentralized: each pod is self governed and limited rules apply to the system. A loosely defined system allows data publisher to be opinionated.
- Query abstraction layer: we want the abstraction to happen through a declarative query. In this work we use the SPARQL query language.

1.4 Hypotheses

Our hypothesis is that we can create an automated client capable of deciding where to store a resource given a pod. We hypothesize that the overhead such an intelligent client has, in comparison to a client that is not smart, is limited. Concretely, we expect a maximum execution time overhead of four times, and maximum double the HTTP requests. For applications that do not write too often, this is an acceptable overhead for the amount of complexity it takes away from developers. Even more so, write speeds are, in contrast to read speeds, typically not critical, since users often don't need them for interactivity. The reason being that applications are typically created in such a way that they synchronize local changes in the background, without disturbing the user.

1.5 Outline

In the next chapter, we describe the Semantic Web, an idea launched by the inventor of the Web almost two decades ago. The semantic web is an enormous research domain, so we limit the discussion to what is required to understand this work. We therefore focus on data representation, data query language, data validation and data retrieval.

We continue by discussing parts of the Solid specification and positioning it in the broader data storage field.

In our fourth chapter, we shortly discuss the use case used throughout the examples of this work.

After discussing the use case, we discuss our contributions in detail. To clearly explain the introduced vocabulary, we start by providing a high-level overview, after which we walk through some user flows. Only once those high-level views are explained will we delve into the details of the vocabulary. We conclude the chapter on our contributions by sketching three short configurations that handle a single use case.

After explaining vocabulary, we continue to evaluate it. This evaluation knows both a theoretical and empirical faced. In the end, we will conclude that our hypothesis does indeed hold.

Now that we proved our work heads in the right direction, we look onto to the future, discussing various research opportunities. Since this work is, to our knowledge, the first of its kind in this ecosystem, we see a lot of research avenues.

We end in a conclusion where we look back at the presented work.

2 Semantic Web

The Semantic Web is a W3C (*World Wide Web Consortium*) initiative that aims to extend the human-readable web to a machine-readable web. The initiative started in 2001 by the inventor of the web, Sir Tim Berners-Lee [25], and has grown to be a mature technology. Even though the technology is mature, it is not outdated, with new specifications still being created to keep the system up to date with today's requirements. This chapter aims to give a high-level overview that is limited to the technologies used in this work.

2.1 RDF

RDF (*Resource Description Framework*) [8] is a W3C specification that models graph data using triples. A triple `<s, p, o>` contains a subject, predicate, and object. Each element of the triple can be a URI and can thus be dereferenced using an HTTP GET request. A dereferenced URI should contain additional info about that subject. A triple can thus be modelled as an arrow labelled with a predicate from subject to object, and each of these is a node that describes itself. Objects can also be literal values like strings, integers, etc.

Subjects and objects can be blank nodes, these nodes are only addressable in the context of the same file. This means that a triple can contain a blank node, which cannot be dereferenced using HTTP. The info related to that blank node is contained within the same document the triple resides in.

A triple exists in the context of a named graph, and when no graph is provided, the triple exists in the `defaultgraph`. When adding a graph to each triple, we define an RDF entry as a quad: `<s, p, o, g>`. In this work, we will always work in the default graph as a simplification. We can make this simplification because Solid does not rely on graphs, and adding them would be a nuance.

2.1.1 Serializations

RDF can be serialized using different formats. The formats used in this work are the machine format n-triples and the human format turtle, but many more exist.

N-Triples

The N-Triples [26] format is an unordered serialization, serializing each triple separated by a dot. The symbols `<>` are used to denote a URI. Values not contained within `<>`

```
<http://base.example.com/me> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://base.example.com/me> <http://xmlns.com/foaf/0.1/givenName> "Alice" .
<http://base.example.com/me> <http://xmlns.com/foaf/0.1/familyName> "Rabbit"^^<http://www.w3.org/2001/
XMLSchema#string> .
<http://base.example.com/me> <http://xmlns.com/foaf/0.1/knows> <http://example.org/Bob> .
<http://base.example.com/me> <http://xmlns.com/foaf/0.1/knows> <http://example.org/Carol> .
<http://base.example.com/me> <http://xmlns.com/foaf/0.1/knows> _:ub2bL9C5 .
_:ub2bL9C5 <http://xmlns.com/foaf/0.1/givenName> "Dave" .

<http://example.org/Bob> <http://xmlns.com/foaf/0.1/givenName> "Bob" .
<http://example.org/Bob> <http://xmlns.com/foaf/0.1/familyName> "Builder" .
```

Listing 1: An example N-Triples document

are considered either blank nodes or literal values. Blank nodes are represented by a "_:" prefix followed by an identifier. The format of a literal is first the value of the literal between double quotation marks (`""`), followed by "`^^`" and then the data type. When no data type is given, the string data type (`<http://www.w3.org/2001/XMLSchema#string>`) is assumed. Listing 1 shows an example N-Triples serialization.

Turtle

The turtle file format is an extension to N-triples, specifically designed to be easier to read for humans [27]. It introduces prefixes to reduce the size of each triple, increasing readability. Each turtle triple is ended using either ";", "," or ".", depending on the chosen character, your triple shares, respectively, the object, the object and predicate, or nothing with the previous triple. Blank nodes have some additional syntactic sugar and can be created using brackets ("`[]`") in which the predicate and object belonging to the blank node are contained. An additional feature is the use of a "base". Turtle allows you to specify named nodes (URIs) in relation to a base by using the `<>` containing a path instead of a whole URI. The base URI should be provided when parsing the turtle file. Listing 2 is a turtle serialization of Listing 1 when using the base URI "http://base.example.com/".

2.1.2 Concise Bounded Description

The CBD (*Concise Bounded Description*) [4] of a RDF resource is the set of triples that can be created as follows:

1. Create a `to-visit` set equal to the set of triples that has the focussed resource as a subject

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<me> a foaf:Person ;
  foaf:givenName "Alice" ;
  foaf:familyName "Rabbit" ;
# Two triples sharing the same subject and predicate
  foaf:knows ex:Bob, ex:Carol ;
# A blank node
  foaf:knows
    [
      foaf:givenName "Dave"
    ] .

ex:Bob
  foaf:givenName "Bob" ;
  foaf:familyName "Builder" .
```

Listing 2: An example Turtle document

2. Iterate over the triples in the to-visit set and:
   i. Add the current triple to the result set.
   ii. In case the object of the current triple is not a named node: add all triples with that object as a subject to the to-visit set.
   iii. Remove the current triple from the to-vist set.

In this work we often use "the RDF resource" to refer to the CBD. As an example, Listing 3 shows the CBD of Listing 2.

2.2 SPARQL

The SPARQL query language is a declarative query language like, for example, SQL, but specifically designed for RDF [28]. The query language is very extensive, in this section we explain what is needed to understand this work.

SPARQL and turtle share a lot of syntax, with a minor nuance in prefix declaration. Turtle uses @PREFIX to define a prefix, while SPARQL just uses PREFIX. Turtle also expects a dot at the end of a prefix declaration, while SPARQL does not. Additionally,

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<me> a foaf:Person ;
  foaf:givenName "Alice" ;
  foaf:familyName "Rabbit" ;
  foaf:knows ex:Bob, ex:Carol ;
  foaf:knows [
      foaf:givenName "Dave"
    ] .
```

Listing 3: The Concise Bounded Description of <me> from Listing 2

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

Listing 4: An example SPARQL select query

a query specifies the operation used, like SELECT. Listing 4 shows an example SPARQL select query.

2.2.1 Variable

To select a part of a triple, you use a new variable, denoted by a `?` or `$` prefix. The variable "id" would be referenced using `?id`. The variable can then be used in a where clause. The result of a SPARQL select query is a list of bindings that satisfy the projection of the data through the query.

2.2.2 Functions

Within a query, functions can be used to transform data. We mention the functions used in this work.

Bind

Binds a certain value, or variable, to a variable. To bind the value "apple" to the variable "pear", you would use: `BIND ("apple" as ?pear)`.

STR

The STR function gets the raw string representation of a value, for example when we have a date: `"2024-05-08T23:23:56.83Z"^^xsd:dateTime` we can get the value between double quotation marks by using the STR function. `STR("2024-05-08T23:23:56.83Z"^^xsd:dateTime)` would evaluate to `"2024-05-08T23:23:56.83Z"`.

2.2.3 Property Paths

Property paths allow you to describe a route between two nodes. In this work, we use the `*` property path, which means, following a property/ predicate zero or more times. To express that we want to bind the variable "location" to each geographic location in which our variable "city" is located, we could use: `?city ex:locatedIn* ?location`.

2.2.4 Different kind of queries

Until now, we kept it easy by focussing on read queries. Since this work focuses on write queries, we quickly go over all the different syntaxes SPARQL provides to update a resource.

Insert Data

An insert data query adds some triples listed to a data source. The operation is structured as INSERT DATA { … } replacing the ellipsis with turtle formatted triples.

Delete Data

A delete data query simply removes the triples listed from a data source. If a triple that should be deleted is not present, it is ignored. The operation is structured as DELETE DATA { … } replacing the ellipsis with turtle formatted triples.

Delete / Insert Where

A delete insert query consists of an optional delete clause followed by an optional insert clause, followed by a where clause. Either the delete or insert clause, or both, need to be present. When both are present, the query will have a structure like: DELETE { … } INSERT { … } WHERE { … }. Unlike the “insert data” and “delete data” queries, these queries can contain variables. Important to note is that the where clause is evaluated only once. The resulting bindings are then substituted in both delete and insert clauses, and afterwards the delete clause is executed followed by the insert clause.

Delete Where

A “delete where” query is an abbreviated form of the above query. The query DELETE WHERE { content } is equivalent to DELETE { content } where { content }.

2.3 Shape Descriptions

RDF is what they call a schemaless data format, meaning it does not a priori require a format in which the data will be stored. Other examples of schemaless data are a MongoDB[3] and Redis[4]. Data with a schema are, for example, your traditional relational databases. An in-depth comparison between schema and schemaless data storage is beyond the scope of this work.

[3]https://www.mongodb.com/
[4]https://redis.io/

Schemaless data can after the creation still be validated against some **schema**, this is useful for application developers that expect the data they consume to be in a specific format. In this work, we will use schemas to group similar data together. Within the RDF ecosystem, two schema languages are important, ShEx and SHACL. ShEx was created out of a community need to describe shapes and has a compact syntax. SHACL was created later as a W3C recommendation. This work primarily uses SHACL because it is a W3C recommendation, we assume it will be more future-proof. We will sporadically use ShEx in this text for its compressed format.

2.3.1 ShEx

A ShEx [10] shape essentially lists properties and their accompanying object type, as well as the cardinality of that property in relation to the focussed subject. Listing 5 shows a self-explanatory shape example taken from the ShEx website[5].

2.3.2 SHACL

SHACL [9] is extensively used in this work, yet because the different SHACL properties are rater self-explanatory, we do not give an in-depth explanation on SHACL.

2.4 Interfaces

An interface is the shared boundary between two systems. We are interested in the boundary between a data owner and a data consumer. In other words, say a data owner has some RDF data, how does a data consumer gain access to that data. In this work, we will focus on web interfaces. Numerous RDF interfaces exist, a data owner could choose to expose all data in a compressed format, use a SPARQL endpoint, use TPF (*Triple Patter Fragments*) [12], etc. Another possibility, is to group RDF triples together in dif-

```
# our EmployeeShape reuses the FOAF ontology
<EmployeeShape> {              # An <EmployeeShape> has:
   foaf:givenName  xsd:string+,   # at least one givenName.
   foaf:familyName xsd:string,    # one familyName.
   foaf:phone      IRI*,          # any number of phone numbers.
   foaf:mbox       IRI            # one FOAF mbox.
}
```

Listing 5: Example ShEx shape taken from their website

[5]https://www.w3.org/community/shex/

```
SELECT ?s ?p ?o WHERE {
  ?s ?p ?o
}
```

Listing 6: SPARQL query to select all triples

ferent HTTP documents and provide an interface that links those documents together accordingly. This essentially creates an endpoint following the REST architecture.

2.4.1 SPARQL endpoint

A SPARQL endpoint is a conceptually simple interface. You ask a SPARQL query to the interface and you get the result. As an example, Listing 6 shows how to get all data behind a SPARQL endpoint. Behind the conceptually simple interface is a huge amount of technical complexity. This technical complexity together with a potentially large computational cost of a SPARQL endpoint makes it unfit for some use cases.

2.4.2 LDP

LDP is a set of rules that allow you to create a simple RESTful interface mimicking an operating system's file structure. Within an LDP interface, each HTTP resource returns RDF triples that either describe some resource, or describe some collection that contains other RDF resources. Listing 7 shows an example LDP container. An LDP interface allows CRUD operations through the HTTP methods.

2.5 Query Engines

Query engines are complex pieces of software that answer queries, like for example the SPARQL queries seen in Chapter 2.2. The features supported by the engine are engine-dependent, but they can be extensive. Besides feature support, they can also perform query optimizations based on things like cardinalities that are either known from the start, or are discovered during the query process . A query engine aims to shield the developers from the complexities that are omnipresent when querying data.

```
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix ldp: <http://www.w3.org/ns/ldp#>.
<http://example.org/c1/>
   a ldp:BasicContainer;
   dcterms:title "A very simple container";
   ldp:contains <r1>, <r2>, <r3>.
```

Listing 7: LDP container example

These complexities range from different data formats, to different interfaces, to possible optimizations.

The Comunica query engine [29] has been especially designed to be modular, allowing easy extensibility in the different areas mentioned above. This work will use that engine because of its modular design, existing feature richness and free software nature.

3 Solid

The Solid project develops a specification that lets individuals and groups store their data securely in decentralized data stores called Pods [17]. Pods are like secure web servers for data. When data is stored in a Pod, its owners control which people and applications can access it. Solid is a specification [30] that builds on top of existing specifications. The data is stored in RDF format and the interface is constructed through LDP. However, the existing specifications are not enough as solid faces numerous challenges because of its innovative decentralized nature. These challenges span across multiple domains like interface design, query engine design, access control, usage control, etc. To tackle these challenges, Solid creates some own specifications, but tries to keep them generic for different use cases.

3.1 Positioning

Throughout this work, we approach the collection of all Solid data stores (pods) as a permissioned decentralized graph database. It's important to note that Solid differentiates itself from typical distributed database systems in various ways. A distributed database will both replicate and shard its data [31]. Replication means that the same data is stored on multiple machines, and sharding means that one machine does not hold all the data. We can thus view the data in a distributed database as a collection of shards, these shards are replicated a configurable amount of times and stored across different machines. The replication of data can happen in different configurations, each with their own considerations (the interested reader can read R. Elmasri [31], specifically section 24.1.2). One example consideration is the leader-follower configuration where each update is performed on the leader while reads are performed on both leader and followers. The leader is responsible for synchronizing the data updates to the followers. Such a configuration chooses for availability before consistency on the CAP (*Consistency, Availability, Partition tolerance*) scale because it is possible reads are outdated and thus inconsistent. The CAP theorem states that when designing a system, you can only pick two of the three properties {Consistency, Availability, Partition tolerance}.

Interestingly, Solid does not introduce any replication across Solid pods. From a theoretical standpoint, this means that we can view a single Solid pod as a single shard of our database, and each shard being self governed. As a result of not having data replication, the Solid specification does not position itself in the CAP space, choosing neither consistency nor availability.

3.2 Access Control

For each resource, the Access Control describes what actors have access to the resource. The resource access is often aligned with the CRUD operations, so each operation has their own set of actors that can perform the action.

3.2.1 WAC

WAC (*Web Access Control*) [15] is a decentralized cross-domain access control system, providing a way for Linked Data systems to set authorization conditions on HTTP resources using the ACL (*Access Control List*) model using the ACL ontology[6]. WAC differentiates four access modes[7]:

1. acl:Read: Allows access to a class of read operations on a resource, e.g., to view the contents of a resource on HTTP GET requests.
2. acl:Write: Allows access to a class of write operations on a resource, e.g., to create, delete or modify resources on HTTP PUT, POST, PATCH, DELETE requests.
3. acl:Append: Allows access to a class of append operations on a resource, e.g., to add information, but not remove, information on HTTP POST, PATCH requests.
4. acl:Control: Allows access to a class of read and write operations on an ACL resource associated with a resource.

WAC was created with some extensibility in mind. One could use the Access Mode Extensions[8] to define a subclass of a default access mode, like for example acl:read. Interestingly, the specification includes the following warning:

> "Servers are strongly discouraged from trusting the information returned by looking up an agent's WebID for access control purposes. The server operator can also provide the server with other trusted information to include in the search for a reason to give the requester the access."

[6]http://www.w3.org/ns/auth/acl#
[7]https://solidproject.org/TR/wac#access-modes
[8]https://solidproject.org/TR/wac#extension-acl-mode

```
@prefix acl: <http://www.w3.org/ns/auth/acl#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix alice: <https://example.com/Alice#> .
@prefix bob: <https://example.com/Alice#> .

[
  acl:accessTo alice:card ;
  acl:mode acl:Read ;
  acl:agentClass foaf:Agent
] .
[
  acl:accessTo alice:card ;
  acl:mode acl:Read, acl:Write ;
  acl:agent bob:card
] .
```

Listing 8: Example WAC description

This is interesting in the context of solid because it means we are discouraged from making access rules like: “Access is granted when the requestor is older than 21.” This consideration was possibly written down to warn readers that servers cannot validate the data. However, verifiable credentials might be able to change this view. Unfortunately, a WAC GitHub issue about this is open and inactive[9]. Listing 8 shows an example WAC description.

3.2.2 ACP

ACP (*Access Control Policy*) [2] is an alternative to WAC and serves as the older sibling. ACP allows you to create matchers over users. These matchers can return a value true or false based on the agent that requests the resource. Policies are used to connect matchers to resources, and the access modes used are the same as WAC. We provide an ACP example in Listing 9.

Altough ACP is more expressive than WAC, neither of the solutions are perfect. They both lack the true meaning behind a policy. ACP requires you to provide a rule for each resource, but has no way to generalize resources. I might, for example, want to create an access control resource that grands Alice access to the subset of my pictures that they are contained in. Since rules relate to a specific resource, ACP and WAC lack the ability to express this.

[9]https://github.com/solid/authorization-panel/issues/79

```
ex:accessControlResourceA
  acp:resource ex:resourceX ;
  acp:accessControl [
    acp:deny acl:Read, acl:Write ;
      acp:anyOf [
        acp:client acp:PublicClient ;
      ] ;
      acp:noneOf [
        acp:client ex:clientC
      ] ;
  ] .
```

Listing 9: Example ACP description

3.3 Usage Control

Beyond access control, the Solid Community is increasingly investigating usage control solutions. Usage Control takes access control that decides whether you have access to a resource, and expands on it by describing how you can get access [32]. Additionally, it describes what you can do with the resource after access has been granted. These permissions are related to the deontic concepts: Permission, Prohibition, Obligation and Dispensation.

3.4 Pod Descriptions

A Solid pod following the current specification has an LDP interface. Such an interface is unstructured by design, forcing a data consumer to traverse all links in the same pod to get a complete pod overview. If completeness is of importance, this makes an LDP interface worse than a bulk download. To avoid this, a pod can have an index that can be used to speed up query execution. When creating an index, special care should be given to not leak information about the data stored in the pod that the requestor would not have access to.

3.4.1 Type Index

The first index proposed for Solid was the Type Indexes specification [22]. It suggests two indexes, a private and a public index. Each index contains entries that map a certain RDF type to a set of HTTP resources. Listing 10 shows a type index that states that RDF resources that have a tuple like `<s rdf:type vcard:AddressBook>` can be found at path `/public/contacts/myPublicAddressBook.ttl`.

```
@prefix solid: <http://www.w3.org/ns/solid/terms#>.
@prefix vcard: <http://www.w3.org/2006/vcard/ns#>.
@prefix bk: <http://www.w3.org/2002/01/bookmark#>.

<>
  a solid:TypeIndex ;
  a solid:ListedDocument.

<#ab09fd> a solid:TypeRegistration;
  solid:forClass vcard:AddressBook;
  solid:instance </public/contacts/myPublicAddressBook.ttl>.
```

Listing 10: An example type index

Besides the low granularity type indexes allow, they are inherently flawed because the access of resources cannot be grouped into “public” and “private” since more complex access control policies are the norm.

3.4.2 Shape Tree

Shape Trees [23] are the proposed replacement for the Type Indexes. The specification uses shape descriptions like ShEx and SHACL to validate RDF graphs against a set of conditions. Shape trees can be used in combination with protocols that organize Linked Data graphs into resource hierarchies, expressing the layout of the resources and associating those resources with their respective shapes. It is the natural extension of shape descriptions to those resource hierarchies.

The shape tree specification can be used on top of any technology platform that supports the notion of containers and resources, but it is mostly used on top of LDP. The shape tree specification defines a predicate st:contains that asserts a “physical” hierarchy. The “physical” containment is defined as LDP containments. The shape tree specification also defines virtual containment, this is just another way of realizing directories above the underlying LDP specification. It means you do not need ldp:contains for defining containers, but can define another predicate, and use that predicate to create directories. Essentially, it makes you able to view ex:apple1 and ex:apple2 as containing resources of ex:appleTree as seen in Listing 11.

By creating a graph of shape descriptions, access control using shape trees has a finer granularity compared to Type Indexes. Each Subtree can be exposed through its HTTP resource and can therefore have their own access control policies. The privacy of a user can thus be protected by exposing a shape tree in a small documents.

```
@prefix ex: <http://example.org/> .
ex:appleTree
  ex:hasFruits ex:apple1, ex:apple2.
```

```
<#VirtualAppleTree>
  a st:ShapeTree ;
  st:expectsType st:Resource ;
  st:shape ex:AppleTreeShape ;
  st:references [
    st:referencesShapeTree <#VirtualAppel> ;
    st:viaPredicate ex:hasFruits
  ] .

<#VirtualAppel>
  a st:ShapeTree ;
  st:expectsType st:Resource ;
  st:shape ex:AppleShape ;
```

Listing 11: Shape tree example: resource (left) can be described by a shape tree (right)

3.5 Solid Interoperability

The Solid Interoperability specification [33] is a big specification that essentially ties together the smaller specifications and offers some usage patterns. It essentially describes how applications should work together to ensure a coherent data ecosystem.

The specification differentiates social agents (individual, group, or organization) and applications, considering both to be *agents*. Social agents choose to use certain applications and register/ manage them and their access rights in a private document. It considers data to be stored in data registries, each indexed using a shape tree, and describes that resource names should be unpredictable. The unpredictability is important for the privacy of data owners. The specification also describes how one can request access to a resource and a way of storing what access rights have been granted.

4 Use Case

In this work, we will work with a social media use case where a pod contains the social media data of a single user. This use case follows the LDBC SNB (*Social Network Benchmark*) [11] use case. Every person has information about themselves and can create *posts* and *comments*. Both posts and comments are *messages*, and a comment is a reply to some message. Messages have an ID, a browser, a location IP, content, tags, and a creator. Figure 6 shows the schema used within this use case as copied from the LDBC SNB[10] GitHub. Listing 12 and Listing 13 show two example queries over the SNB data that respectively read, and write data.

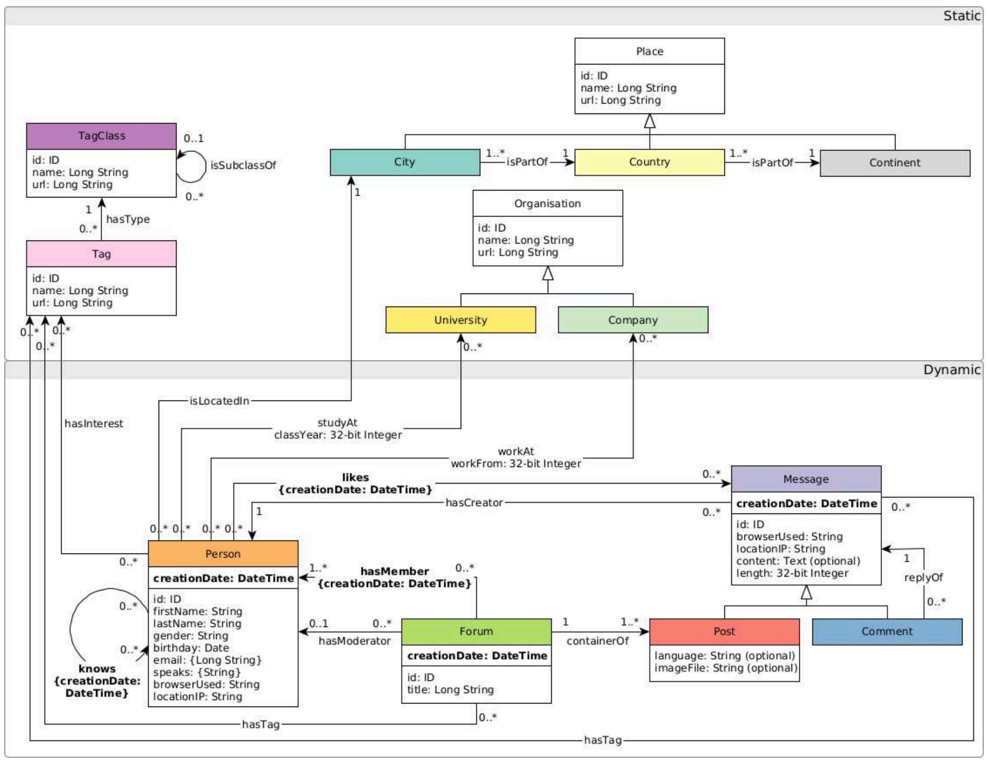

Figure 6: Social Network Benchmark data schema

[10]https://github.com/ldbc/ldbc_snb_datagen_hadoop#graph-schema

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/>
SELECT ?firstName ?lastName ?birthday ?locationIP ?browserUsed ?cityId ?gender ?creationDate
WHERE {
  <https://example.com/Alice/profile/card#me> rdf:type snvoc:Person;
    snvoc:id ?personId;
    snvoc:firstName ?firstName;
    snvoc:lastName ?lastName;
    snvoc:gender ?gender;
    snvoc:birthday ?birthday;
    snvoc:creationDate ?creationDate;
    snvoc:locationIP ?locationIP;
    snvoc:isLocatedIn ?city.
  ?city snvoc:id ?cityId.
  <https://example.com/Alice/profile/card#me> snvoc:browserUsed ?browserUsed.
}
```

Listing 12: Example LDBC SNB read query

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix tag: <http://localhost:3000/www.ldbc.eu/ldbc_socialnet/1.0/tag/>
prefix resource: <http://localhost:3000/dbpedia.org/resource/>

INSERT DATA {
  <http://example.com/Alice/Posts#416608218494388> a ns1:Post ;
    ns1:browserUsed "Chrome" ;
    ns1:content
      "I want to eat an apple while scavenging for mushrooms in the forest." ;
    ns1:creationDate "2024-05-08T23:23:56.830000+00:00"^^xsd:dateTime ;
    ns1:id "416608218494388"^^xsd:long ;
    ns1:hasCreator <http://example.com/Alice/profile/card#me> ;
    ns1:hasTag tag:Alanis_Morissette, tag:Austria ;
    ns1:isLocatedIn resource:China ;
    ns1:locationIP "1.83.28.23" .
}
```

Listing 13: Example LDBC SNB write query

5 Storage Guidance Vocabulary

To empower automated clients to correctly store RDF resources, we suggest the usage of a descriptive vocabulary. Existing structure definitions of data spaces like Type Index [22] and Shape Trees [23] focus on read queries and insufficiently support write queries. These structure definitions fail to express the underlying decision-making of why a resource is stored where it is. As an example, we give a small list of questions that cannot be answered by either data store descriptions:

1. What if multiple directories match? Do I duplicate the resource?
2. What should I do if no documents match?
3. How are resources grouped?
   i. Can I infer that resources grouped by a property are always grouped by that property?
   ii. Does that mean that if I get a new object for that property that I can just create a new document?
4. What should I do when I update a resource?
   i. Should I alter the data store description?
   ii. Should I move the resource? (Assign a new named node?)
5. Are all clients equal? Do they all abide the structural information description?

To answer these questions, we develop a new vocabulary, namely, SGV (*Storage Guidance Vocabulary*). This vocabulary takes inspiration from the Shape Tree Specification, but does not extend it. The vocabulary aims to express where a resource is stored and why. SGV is created with a primary focus on LDP [7] interfaces, extending LDP containers to be structured. A container marked as structured has a strict definition of where containing containers/resources are located. We shortly introduce some basic concepts in SGV:

Resource Collection: Corresponds to a group of RDF resources.

Unstructured Collection: Corresponds to a classical LDP container or HTTP resource

Structured Collection: A canonical or derived collection. (below)

Canonical Collection: A resource collection containing resources.

Derived Collection: A resource collection that stores resources already stored by one or more other structured containers.

Resource Description: A way of describing resources, for example through ShEx or SHACL.

Group Strategy: A description of how resources should be grouped together, for example: my images are grouped per creation date.

Store Condition: When multiple collections are eligible to store a resource, the store condition decides what collection(s) actually store the resource. Allowing the creation of a store priority system.

Update Condition: Describes what to do when a containing resource is changed.

Client Control: Describes the amount of freedom a client has when trying to store a resource.

We will first describe two simple flows, the creation, and the modification of an RDF resource. This should provide an idea of what SGV tries to accomplish without going into all the details first. After explaining the two example flows, we will look into the details of SGV.

5.1 Flow: A client wants to create an RDF-resource

Inserts happen on a pod level, meaning you specify to the client what pod you'd want to insert a resource to. The client will then discover a fitting location for the resource. Listing 14 is an example query that would trigger the resource creation flow.

An automated client is now required to discover the base (`<>`) of this query. The client will follow the flow described below and visualized in Figure 7.

1. The client gets the SGV description of the storage space (can be cached).
2. The client checks all canonical collections and checks if the resource to be inserted matches a resource description of the collection.
3. If the resource matches a description, the client checks the store condition of the description given the eligible collections.
4. For each collection that stores the resource:
    i. The client checks the group strategy of the collection and groups the resource accordingly, deciding on the name of the new resource.

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix card: <http://localhost:3000/pods/00000000000000000096/profile/card#>
prefix tag: <http://localhost:3000/www.ldbc.eu/ldbc_socialnet/1.0/tag/>
prefix resource: <http://localhost:3000/dbpedia.org/resource/>

INSERT DATA {
 <> a ns1:Post ;
   ns1:browserUsed "Chrome" ;
   ns1:content
     "I want to eat an apple while scavenging for mushrooms in the forest." ;
   ns1:creationDate "2024-05-08T23:23:56.830000+00:00"^^xsd:dateTime ;
   ns1:id "416608218494388"^^xsd:long ;
   ns1:hasCreator card:me ;
   ns1:hasTag tag:Alanis_Morissette, tag:Austria ;
   ns1:isLocatedIn resource:China ;
   ns1:locationIP "1.83.28.23" .
}
```

Listing 14: Example resource insertion query

  ii. The client checks the collections that are derived from this collection. Step 4 is executed for all collections that are derived from this collection, and the resource matches the description.

5. The client performs the store operation.

5.2 Flow: A client wants to update an RDF-resource

An update can be both an insert to an existing resource, a change in values of a resource, or a deletion of the whole, or part of a resource. In case of an update, it's important that the client knows what resource will be updated. This is similar to how queries are executed right now, where you should always specify the HTTP resource to query over (excluding link-traversal clients).

The flow of an automated client is depicted in Figure 8 and described further below.

1. The client gets the SGV description of the storage space and the HTTP resource containing the updated RDF resource.
2. The client virtually constructs the resource that would result from the requested operation.
3. The client check the update condition of the original matching resource description. Following action depends on the update condition. Typically, the update-condition will say whether an RDF resource is moved or not.

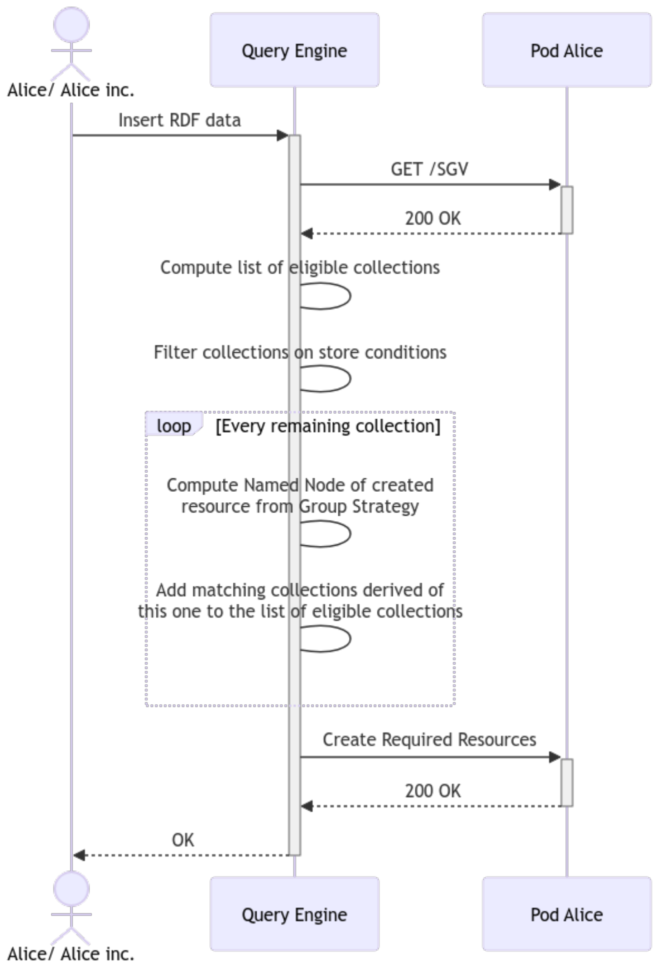
Alice/ Alice inc.
Query Engine
Pod Alice
Insert RDF data
GET /SGV
200 OK
Compute list of eligible collections
Filter collections on store conditions
loop [Every remaining collection]
Compute Named Node of created resource from Group Strategy
Add matching collections derived of this one to the list of eligible collections
Create Required Resources
200 OK
OK
Query Engine
Pod Alice
Alice/ Alice inc.

Figure 7: Flow: create RDF resource

i. Move required: remove the existing resource and follow the steps described in Chapter 5.1.

ii. No move required: just update the resource as requested by the user.

5.3 Details

This section delves into the details of SGV. The vocabulary is constructed with future expansions in mind, missing features could thus be added by other actors. In Figure 10 an overview of the different components can be consulted. The figure can be used as a reference while reading the different sections. There are three arrows used in the graph, each with a different meaning, visualized in Figure 9. Firstly, a full arrow means that there can be a triple `?a ldp:contains ?b`. Secondly, the dotted arrow means that the destination has the same fields or more as the source. Finally, a diamond shaped arrow entails a link from the source to the destination, specifically, the destination can be considered a property of the source. Listing 15 is provided as an example description

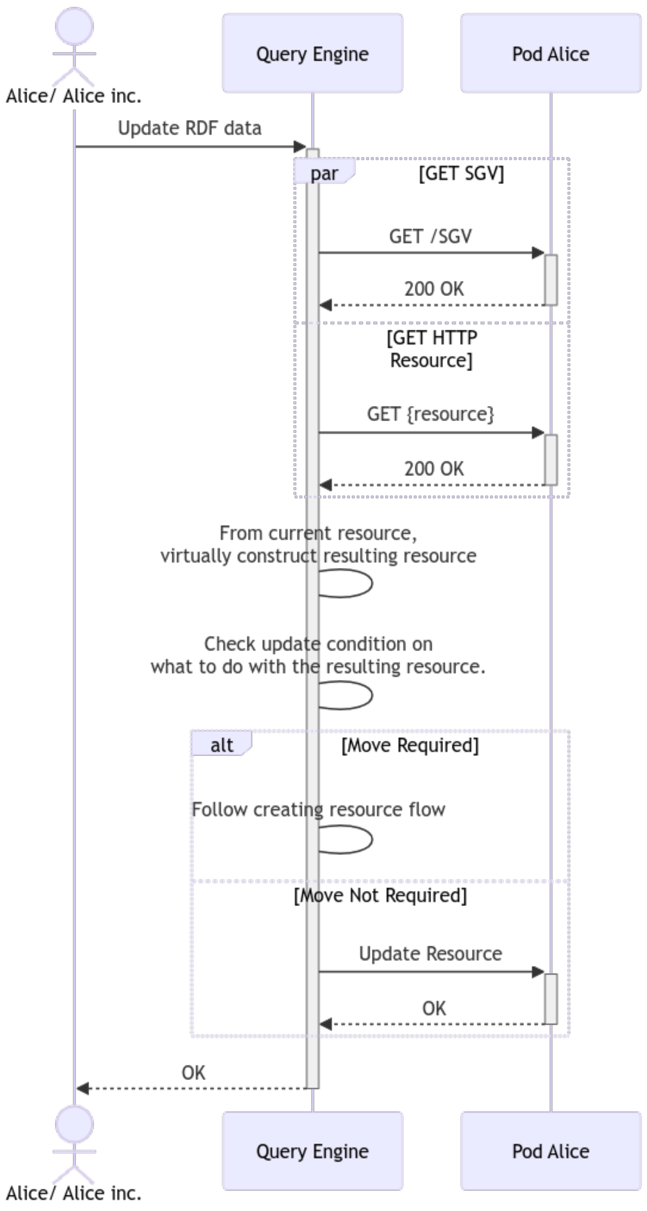

Figure 8: Flow: update RDF resource

to help clarify the vocabulary. For completeness, we also provide the ShEx description of the vocabulary in Listing 16.

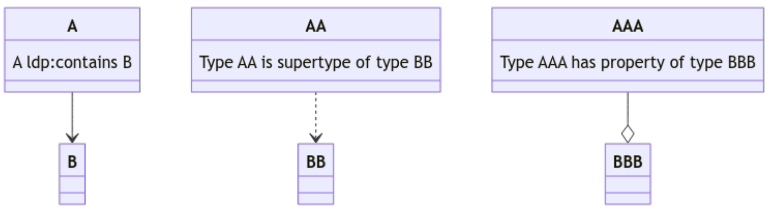
A
A ldp:contains B
B
AA
Type AA is supertype of type BB
BB
AAA
Type AAA has property of type BBB
BBB

Figure 9: A legend explaining the links used in Figure 10

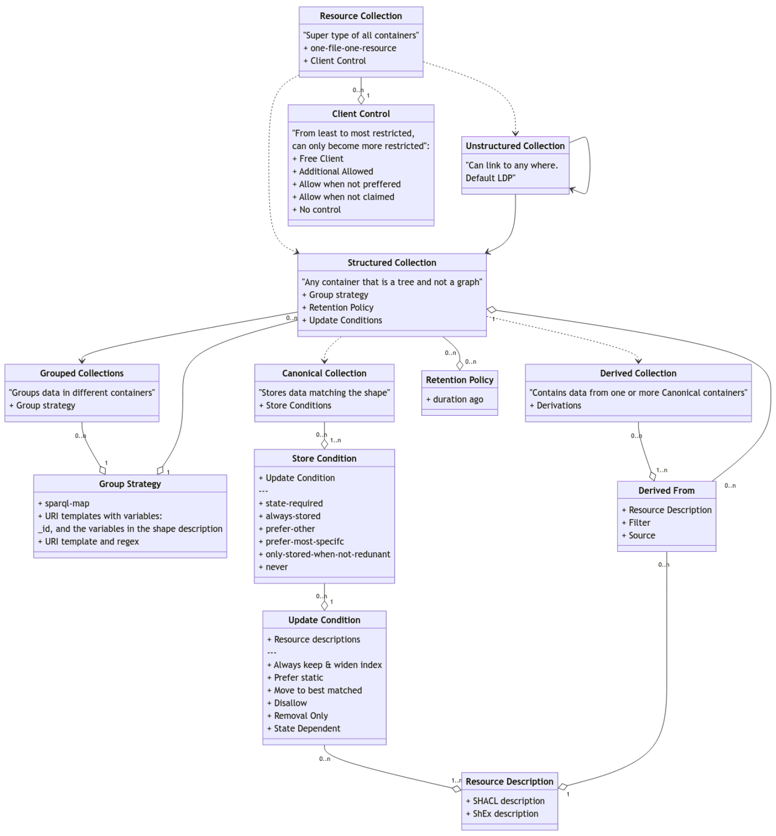

Figure 10: Visualisation of the Storage Guidance Vocabulary

5.3.1 Resource Collection

An sgv:resource-collection is any RDF resource that groups multiple RDF resources together. This grouping into a collection can be done in either an explicitly structured or unstructured way. Note that we group RDF resources, a collection can either be an LDP Container, or an HTTP resource.

5.3.2 Unstructured Collection

An unstructured collection is a kind of resource collection (Section 5.3.1), that does not explicitly define its structure. This work's primary focus is to enable automated clients to perform insert queries over LDP interfaces. It might help to see the type sgv:unstructured-collection as similar to the ldp:Container type.

5.3.3 Structured Collection

An sgv:structured-collection is a resource collection (Section 5.3.1) that explicitly describes its structure. The collection defines a filter, each resource is compared against this filter. If

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex: <http://example.org/> .
@prefix sgv: <https://example.org/storage-guidance-vocabulary#> .
@prefix ldp: <http://www.w3.org/ns/ldp#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ldbc: <http://www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/> .
@prefix dbo: <https://dbpedia.org/ontology> .

<> a ldp:Container, sgv:unstructured-collection ;
 sgv:client-control [
    a sgv:allow-when-not-claimed ;
  ] ;
 sgv:one-file-one-resource "false"^^xsd:boolean .

# An unstructured collection contains a structured collection "posts"
<posts/> a ldp:Container, sgv:structured-collection, sgv:canonical-collection ;
 sgv:one-file-one-resource "false"^^xsd:boolean ;
 sgv:store-condition [
    a sgv:always-stored ;
    sgv:update-condition [
       a sgv:update-prefer-static ;
       sgv:resource-description [
         a sgv:shacl-descriptor ;
         sgv:shacl-shape <sgv#postShape> ;
       ] ;
     ] ;
  ] ;
 sgv:group-strategy [
    a sgv:group-strategty-uri-template ;
    sgv:uri-template
 '{http%3A%2F%2Fwww.ldbc.eu%2Fldbc_socialnet%2F1.0%2Fvocabulary%2FisLocatedIn}#{::UUID_V4}' ;
    sgv:regexMatch '([^/]+)#([^#]+)$' ;
    sgv:regexReplace '$1/$2' ;
  ] .

<sgv#postShape>
 a sh:NodeShape ;
 sh:property [
    sh:path rdf:type ;
    sh:hasValue ldbc:Post ;
  ] ;
 sh:property [
    sh:path ldbc:creationDate ;
    sh:datatype xsd:dateTime ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] ;
 sh:property [
    sh:path ldbc:id ;
    sh:datatype xsd:long ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] .
```

Listing 15: Example pod description using Storage Guidance Vocabulary

a resource passes, the resource is stored into the collection. The collection later defines where the resource should be stored. Where a normal resource collection can contain resources in a graph structure, a structured container adds the important restriction

```
PREFIX sgv: <https://thesis.jitsedesmet.be/solution/storage-
guidance-vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-
ns#>
PREFIX ldes: <https://w3id.org/ldes#>
PREFIX tree: <https://w3id.org/tree#>

<ResourceCollectionShape> {
   rdf:type [ sgv:resource-collection ] ;
   sgv:client-control {
      rdf:type [
      sgv:free-client sgv:additional-allowed
      sgv:allowed-when-not-preffered
      sgv:allow-when-not-claimed sgv:no-control
      ] ;
   } ? ;
   sgv:one-file-one-resource xsd:boolean ? ;
}

<StructuredCollectionShape> {
  &<ResourceCollectionShape> ;
  rdf:type [ sgv:structured-collection ] ;
  sgv:group-strategy <GroupStrategyShape> ;
# Copied from the LDES vocabulary, fan of the idea!
  sgv:retention-policy {
    rdf:type ldes:DurationAgoPolicy ;
    tree:value xsd:duration ;
  } ?
}

<CanonicalCollectionShape> {
  &<StructuredCollectionShape> ;
  sgv:store-condition {
    rdf:type [
      sgv:state-required sgv:always-stored
      sgv:prefer-other sgv:prefer-most-specific
      sgv:only-stored-when-not-redundant sgv:store-never
    ] ;
    sgv:update-condition <UpdateConditionShape>
  } + ;
}
```

```
<DerivedCollectionShape> {
  &<CanonicalCollectionShape> ;
  sgv:derived-from {
    sgv:resource-descripion <ResourceDescriptionShape> ;
    sgv:source IRI ;
    sgv:filter xsd:string ;
  } +
}

# Any ldp:Container in a structured collection is a
GroupedCollection
<GroupedCollection> {
  &<GroupStrategyShape>
}

<GroupStrategyShape> {
  (rdf:type sgv:group-strategty-uri-template ;
    sgv:uri-template xsd:string ;
    sgv:regexMatch xsd:string ;
    sgv:refexReplace xsd:string ;
  )|(
    rdf:type sgv:group-strategy-sparql ;
    sgv:sparql-query xsd:string ;
)}

<UpdateConditionShape> {
  (
    rdf:type [
      sgv:update-keep-always sgv:prefer-static sgv:best-
matched
      sgv:update-disallow sgv:removal-only sgv:state-
dependent
    ] ;
  |
    rdf:type [ sgv:keep-distance ] ;
    sgv:distance xsd:decimal ;
    sgv:original-description <ResourceDescriptionShape> ;
  ) ;
  sgv:resource-descripion <ResourceDescriptionShape> ;
}

<ResourceDescriptionShape> {
  rdf:type sgv:shacl-descriptor ;
  sgv:shacl-shape IRI ;
}
```

Listing 16: Shape description of Storage Guidance Vocabulary. Starts left, continues on the right.

of a tree. It has been proven that this tree structure limitation heavily restricts the interface [34].

5.3.4 Canonical Collection

A sgv:canonical-collection is a structured collection (Section 5.3.2) that stores RDF resources. When entering a Solid pod, we check what canonical containers want to store the resource. The collections then individually decide where to store the resource, given the other collections that are eligible to store.

5.3.5 Derived Collection

A sgv:derived-collection is a structured collection (Section 5.3.3) that contains (part of) resources contained in one or more other collections. Existing work around derived resources in solid specifies a “template”, “selector” and “filter” [35]. SGV uses those same components but in a different format. The template describes where the resource should be stored, in SGV this is done using the group strategy (Section 5.3.8). The selector describes what resources are derived. In SGV the selector is a combination of the Resource Description and Source in the “Derived from” node. As for the filter or projection, SGV uses a construct query over the RDF resource. This is different from the work of J. Van Herwegen and R. Verborgh where the construct if performed on HTTP resources which contain multiple resources [35]. In case no filter is present, each resource is derived as a whole. A derived collection without a filter defined on a pod with the “one file one resource” flag, can use soft/ hard links, and thus has a very low cost.

When a structured collection inserts, updates or removes an RDF resource, the collections that derive from that collection are informed to act accordingly. A derived collection can be used to create collections and knows a multitude of use cases, some examples are:

- Create a collection of all pictures in my pod, even though I have multiple canonical collections managing pictures.
- Create a restricted view of resources that I could then use to share with others.

5.3.6 Grouped Collection

Every LDP container contained in a structured collection (Section 5.3.3) is a grouped collection. Grouped collections are used to group resources in structured collections together to provide additional structure. This reduces cognitive load when browsing a collection. Figure 11 visualizes two ways of organizing images, on the left by city and depicted person, and on the right by creation date and depicted person.

5.3.7 Resource Description

Both the canonical collection (Section 5.3.4) and the derived collection (Section 5.3.5) require a description of the RDF resources contained. The description is used to filter

```
└── pictures/
    ├── Valencia/
    │   ├── Klaas.ttl
    │   └── Jitse.ttl
    ├── Ghent/
    │   ├── Simon.ttl
    │   └── Hoyyiw.ttl
    └── Paris/
        ├── Jonas.ttl
        ├── Ana.ttl
        └── Liesbet.ttl
```

```
└── pictures/
    ├── 30-01-2024/
    │   ├── Erin.ttl
    │   └── Oscar.ttl
    ├── 14-02-2024/
    │   ├── Henri.ttl
    │   └── Snil.ttl
    └── 17-05-2023/
        ├── Ghent/
        │   ├── Maurice.ttl
        │   └── Lars.ttl
        └── Paris/
            └── Simon.ttl
```

Figure 11: Two Example Group Strategies

resources to be inserted in the pod, and could be used as an index when querying the pod efficiently using link traversal [21]. A structured collection should thus never contain a resource that does not match the shape description. Two popular choices for describing a resource are ShEx and SHACL.

Shape descriptions are powerful and allow expressing complicated expressions including logical constraint components[11] and property pair constraint components[12]. Logical constraint components allow you to perform boolean operations on existing shapes, through: sh:not, sh:and, sh:or, and sh:xone. This allows you to split shapes into parts, these parts could be evaluated once and the result of the evaluation can be shared with other shapes. The sharing of evaluation effectively creates a cache. Property pair constraint components allow you to assert relations between two values present within the same shape.

5.3.8 Group Strategy

A group strategy expresses how a structured collection (Section 5.3.3) should group RDF resources in grouped collections (Section 5.3.6). Every structured container describes one group strategy. A grouped collection can choose to define its group strategy, thereby overruling the group strategy of the structured collection it resides in.

A group strategy maps each RDF resource to part of a URI. The concatenation of the structured collection URI, and the provided part should result in the URI of the resulting resource. A grouped collection with URI https://example.com/pictures/Valencia/ together with a resulting strategy of Alice.ttl would thus result in https://example.com/pictures/Valencia/

[11] https://www.w3.org/TR/shacl/#core-components-logical
[12] https://www.w3.org/TR/shacl/#core-components-property-pairs

Alice.ttl. We suggest two possible ways of grouping resources, closed world through URI-templates [36], or open world through a SPARQL query.

URI Templates

Through URI templates, one can construct a URI based on some context variables. Given the variables `var := "value"` and `hello := "Hello World!"` the URI template `base/{var}/{hello}` would expand to `base/value/Hello%20World%21`. The context available is the CBD [4] of the RDF resource. Since only properties described in the resource description are guaranteed to be present, only those should be accessed in the URI template. The context is referenced by entering the percent encoded [37] representation of the predicate URI. When multiple predicates need to be followed, we separate them using the "/" which is a character that is not present in URI encoded strings. The template `{https%3A%2F%2Fexample.com%2Fowns-house/https%3A%2F%2Fexample.com%2Faddress}` evaluated over Listing 17 would expand to the percent encoded representation of "Front Street 1": `Front%20Street%201`.

Just like we use the "/" to convey special meaning, we also use the ":" to access special variables. We can use the ":" because it is encoded by percent encoding, and is unused by URI-templates. We currently support only one special variable, being "UUID_V4". As an example, the URI template `one-file#{:UUID_V4}` could expand to `one-file#956242de-2c18-4985-8e9e-d490bc8f97b6`.

URI templates by themselves do not allow very complex structures. SGV therefore allows you to express a regex replace over the result of the template.

SPARQL Query

The URI template solution above, although simple in use, has the disadvantage that you can only access the RDF resource itself. To accommodate this shortcoming, we also suggest the use of a SPARQL query that can access the world if it pleases. We could, for example, create a SPARQL query that groups pictures in directories based on the

```
@prefix ex: <https://example.com/> .
ex:Alice a ex:Person ;
  ex:owns-house
    [
      ex:address "Front Street 1"
    ] .
```

Listing 17: Simple resource with blank nodes

creation date and the country a picture was taken in, such as `France-23-07-2023`. When we assume an image only contains the city it was made in, we would need to discover the country the city is in.

We suggest a SPARQL query that uses the variable `?key`. This variable is bounded to the (temporary) named node of the RDF resource. The query expects the return of a variable `?value` returning the location of the resource relative to the collection. Listing 18 shows an example grouping SPARQL query.

It's possible the query needs to be evaluated over other sources to discover required information. For example, the query above might find its city/country data through data made available by Wikidata[13]. A federated query allows us to dereference different sources. This could be used as an attack vector if a bad actor creates a collection that contains everything and then uses the SPARQL query to pass through sensitive information to its own endpoint [38]. We therefore suggest that a pod lists trusted sources in some top-level resource. This would mean that query federation happens top level. R. Taelman and R. Verborgh describe many more possible security issues in their paper [38].

5.3.9 Store Condition

The store condition decides when an RDF resource is stored, given all canonical collections (Section 5.3.4) that are eligible to store the resource. Optionally, additional context could be given as input to the store condition. A canonical collection can have multiple store conditions, and each store condition has an update condition (Section 5.3.10) and therefore a resource description (Section 5.3.7). We suggest six store conditions: 1. state

```
PREFIX ex: <http://example.org/>
SELECT ?key ?value where {
  BIND(CONCAT(STR(?name), "-", STR(?date)) as ?value) .
  ?key ex:creationDate ?date;
  ?key ex:location [
    ex:locatedIn* [
      a ex:country ;
      ex:label ?name ;
    ] .
  ] .
}
LIMIT 1
```

Listing 18: Example group strategy SPARQL query

[13]https://www.wikidata.org/wiki/

required, and 2. always stored, and 3. prefer other, and 4. prefer most specific, and 5. only stored when not redundant, and 6. never.

State Required

The state required store condition is a simple SPARQL query over the current Solid pod. The expected returning `?value` variable is coerced to a boolean, if true, the resource is stored in the collection. The store condition allows for flexible additional features like a basic locking mechanism, where a store is only allowed in case a lock is not present.

Always Stored

A basic store condition, we always store the resource.

Prefer Other

This store condition indicates another collection takes precedence to store over this one.

I could, for example, have the canonical collections “family pictures” and “pictures”. Instead of creating a complex shape description for my “pictures” that excludes the shape of “family pictures”, I could just say that the “family picture” collection takes precedence over the “pictures” collection. This example is written out in Listing 19.

Prefer Most Specific

This store condition specifies that this collection would only store in case its resource description is the most specific to the RDF resource in focus. It uses a distance function to measure how good the resource description describes the resource. A distance function could be the inverse of “the number of triples a projection of the resource by the description would cover”.

We clarify using an example. It’s important to note that the example is by no means a “good” distance function, we just wish to mention it is possible. Listing 20 describes a person with their name, alternative name and birthdate. Assume we have

```
@prefix ex: <https://example.com/> .
@prefix sgv: <https://example.com/storage-guidance-vocabulary#> .

ex:Pictures a sgv:canonical-collection ;
  sgv:store-condition
    [
      a sgv:prefer-other ;
      sgv:other ex:FamilyPictures ;
    ] .
```

Listing 19: Example prefer other SGV description

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
ex:Alice a ex:Person ;
  ex:birthName "Alice"@en ;
  ex:alternativeName "Rabbit"@en ;
  ex:birthdate "1865-10-01T00:00:00Z"^^xsd:dateTime .
```

Listing 20: RDF description of a person

the two SHACL resource descriptions listed in Listing 21. After projection, we would get 1 triple for the left description, being:

```
<http://example.org/Alice> <http://example.org/birthdate> "1865-10-01T00:00:00Z"^^<http://www.w3.org/
2001/XMLSchema#dateTime> .
```

The right description would return the two triples listed below. The shape projection on the right results in the most triples after projection and would thus have the lowest distance.

```
<http://example.org/Alice> <http://example.org/birthName> "Alice"@en .
<http://example.org/Alice> <http://example.org/alternativeName> "Rabbit"@en .
```

Only Stored When Not Redundant

The “only stored when not redundant” store condition stores only if no other collection stores the resource. When choosing between two collections that both have this condition, we select the collection with a named node that is alphabetically first. This condition can be used to create some kind of inbox collection containing resources that do not yet have a dedicated collection. The pod owner could then manually go over the

```
@prefix ex: <http://example.org/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:left-shape a ex:Shape ;
  sh:property
    [
      sh:path ex:brithdate ;
      sh:nodeKind sh:Literal ;
      sh:datatype xsd:dateTime ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
    ] .
```

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-
ns#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
ex:right-shape a ex:Shape ;
  sh:property
    [
      sh:path ex:brithName ;
      sh:nodeKind sh:Literal ;
      sh:datatype rdf:langString ;
      sh:minCount 1 ;
    ] ;
  sh:property
    [
      sh:path ex:alternativeName ;
      sh:nodeKind sh:Literal ;
      sh:datatype rdf:langString ;
      sh:minCount 1 ;
    ] .
```

Listing 21: Two open shape descriptions of the person in Listing 20

inbox and create the required collections. This would primarily be the case for power users that want full control of their storage.

Never

The store condition “never” is fairly simple, it means no new resource should be stored in this collection. We use this condition when we want to have a collection that contains resources, but cannot get new resources.

5.3.10 Update condition

When an RDF resource is updated, the update condition with the shape description matching the original resource is consulted. To prevent links from breaking, we also suppose the optional usage of a forward referencing pattern, preventing links to break in clients that are aware of this. So when resource ex:orininal-name is moved to ex:new-name, there will be a tuple that describes just that: ex:original-name sgv:moved-to ex:new-name. Servers could also be made aware of this triple, returning a 301 redirect to ex:new-name. A move procedure works by removing the existing resource and then inserting the resource in the pod using the insert procedure. We propose multiple update conditions: 1. always keep, and 2. keep distance, and 3. prefer static, and 4. best match, and 5. disallow.

Always Keep

When using the always keep update condition, it does not matter how the resource has been manipulated, the resource will stay in the collection. In case the resource description does not match the updated resource, it should be changed in such a way that it matches the original description and the updated resource.

Keep Distance

The “keep distance” update condition works similar to the “always keep” condition, but places a limit on how much a resource description can stretch from its original form. To implement this update condition, we require some distance metric between shape descriptions. When the distance grows too big, the original description is reapplied and resources not matching the description are moved.

To our knowledge, there does not yet exist a distance metric to see how much two shape descriptions differ, only whether two descriptions are contained [39]. An example metric could be inspired by the Levenshtein distance where, we count the number of additions and deletions of a SHACL properties. Let’s say each addition or deletion has a cost of 1. The distance between the shapes in Listing 21 would be three because to go from the left description to the right, three operations are required:

1. Remove the property with a path of ex:birthdate.
2. Add the property with a path of ex:birthName.
3. Add the property with a path of ex:alternativeName.

Prefer Static

The “prefer static” update condition will keep a resource in the current collection as long as the resource matches a resource description of the current collection, and move the resource when it does not.

Best Match

The “best match update” condition will discover the collection the updated resource would be placed in, and moves the resource in case this collection and the current collection are not the same.

Disallow

The “disallow” update condition rejects any update made to the resource.

Removal Only

The “removal only” update condition rejects all updates except the full removal of the resource.

State Dependent

Like the “state required” store condition, this update condition allows you to create a SPARQL query. A return variable of a simple update condition is expected.

5.3.11 Client Control

Each resource collection (Section 5.3.1) can specify the level of freedom a client/ actor has to deviate from the SGV. A few example control policies are discussed: 1. free client, and 2. additional allowed, and 3. allowed when not preferred, and 4. allowed when not claimed, and 5. no control. Important to note, no client control allows a client to enter an invalid state. When a state would be invalid to the SGV description, the client needs to update the description.

Free Client

A collection that specifies a client is free, specifies that the client itself can choose where a resource is stored. Since the collection still needs to be in a correct state, the client might have to edit SGV descriptions. Take again the example of "pictures" and "family pictures" collections, where normally a picture matching the family picture description, would be placed in that collection. A free client might choose to store the resource only in the general pictures collection and not in the family pictures collection. They can choose to do this without changing the SGV description.

Additional Allowed

The "additional allowed" client control describes that the client might decide that a canonical container stores a resource it would normally not. Take the pictures example above, the client might decide to store a family picture in both collections.

Allowed When Not Preferred

The "allowed when not preferred" client control states that a client may decide where to store a resource when no collection explicitly wants to store the resource. The collections that want to explicitly store a resource are those collections that would store a resource, but do not have the store condition (Section 5.3.9) "only stored when not redundant". The idea here is that that store condition is only used for those collections that are a last resort to saving a resource automatically. A client can in this case see this last resort option and perform a more suitable action.

Allowed When Not Claimed

The "allowed when not claimed" client control policy describes that a client may decide where to store a resource in case no collection wants to store the resource. This policy deviates from the "allowed when not preferred" policy because this one does not have a store condition filter.

No Control

The "no control" client control allows no deviation from the SGV rules. If a resource is not stored, it will not be stored.

5.3.12 One File One Resource

A big advantage of LDP is that it easily maps to the file structure storage of a typical file systems. If someone wants to store their Solid Pods on their own machine, it's easy for them to access the data. The one file one resource flag signals an LDP server that no HTTP fragments[14] are present in the named nodes in this collection. The server can therefore use soft-links or hard-links to reduce the physical data duplication.

5.3.13 Retention Policy

As a little extra, SGV could be expanded with retention policies like those present in LDES [6].

5.4 Use Case: No Collection Claims Resource

Now that the details are understood, we sketch 3 cases of handling resources that are not claimed. We propose either notifying the owner, letting the client assume a location, or to deny the operation.

5.4.1 Notification

When the owner would like to receive a notification when a resource is not claimed by their pod, they would create a "SGV notification" collection. That collection would have a resource description that matches any resource and a corresponding store condition of "only stored when not redundant". If the pod owner wants to force a client into this use case, the root resource collection would need a client control to be set to "no control". When the user of the client is also the pod owner, the client could provide the user with a popup requesting to handle the notification immediately.

5.4.2 Assume

In the case that a pod owner would want the client to assume the location, root resource collection would need a policy less strict than "no control". In addition, no notification collection as described above can be present if the client control would be "allowed when not claimed".

5.4.3 Deny

In case the pod owner never wants to store a resource that cannot be stored by their SGV description, the root resource collection would need to have a client control of "no control" and no notification collection should be made.

[14]https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web#fragment

6 Evaluation

This chapter provides an extensive evaluation of SGV introduced in Part 5. To evaluate the vocabulary, we implemented a query engine with a minimal set of features from SGV. After discussing the implementation, we shortly discuss the theoretical cost of our operations. We finish with an empirical evaluation of the query engine.

6.1 Implementation

To analyse the capabilities of SGV, we implemented a query engine capable of parsing a pod's SGV description and acting accordingly. The source code of the implementation and benchmark can be found online[15]. The query engine acts as a wrapper around the modular Comunica query engine [29]. We chose to implement a wrapper around Comunica for convenience because it allows us to quickly get results without the need of understanding, or changing Comunicas internal code.

For this proof of concept implementation, we only support essential parts of SGV. We therefore only provide an implementation of the following concepts:

1. Canonical Collection
2. Group Strategy: only URI templates.
3. Resource Description: only SHACL.
4. Store Condition: "always stored", "prefer other", "only stored when not redundant", and "never stored".
5. Update Condition: "prefer static", "move to best matched", and "disallow".

To parse and validate our ShEx descriptions, we use the rdf-validate-shacl library[16]. This library is known to be quite inefficient and could be replaced by the faster SHACL engine library[17]. Unfortunately, that library does not have type descriptions available, making adoptions less desirable.

6.2 Theoretical Evaluation

In our theoretical evaluation, we analyse the number of HTTP requests. In Chapter 1.4 we hypothesize that the required number of HTTP queries of an SGV aware client would at most be double that of a normal one.

[15]https://github.com/jitsedesmet/sgv-update-engine/releases/tag/v0.0.2
[16]https://www.npmjs.com/package/rdf-validate-shacl
[17]https://www.npmjs.com/package/shacl-engine

6.2.1 Insert Operation

In this section, we analyse the cost of a simple insert operation as seen in Listing 28 in the empirical evaluation. In Chapter 5.1 we analysed the steps required for this operation.

Fetch the Description

The query engine should request the SGV description. This accounts for one HTTP request, assuming the API (*Application Programming Interface*) publishes it as a single HTTP resource. It should be noted that the SGV description can easily be cached since it will not change a lot.

Do note however that using an outdated version can be detrimental. Unfortunately, since LDP exposes a pod through multiple different HTTP resources, there is no way of checking whether your SGV description has not grown outdated when you start updating data. For example, you could compute the change, then fetch the SGV again using an if-moified-since header[18] and recompute in case it did change. However, in between confirming you have the latest value and writing the data, the SGV could have been changed, causing you to write in an outdated way, nevertheless. Because LDP exposes multiple HTTP resources, using the If-Unmodified-Since[19] is not possible.

Loop the Resource Descriptions

The next thing a query engine must do is checking what canonical collections want to store the resource. Worst-case scenario, all collections could store the resource, but they only discover this at the last resource description of each collection. In such a case, all resource descriptions pointed to by canonical collections need to be checked.

The cost of a single validation can be linear in the number of properties the description has. Since the focus of the resource is on a single named node, only that named node should be considered as a focus node in the validation.

The computational load could be reduced when resource descriptions have overlapping descriptions. A shape could in that case be defined as a conjunction using `sh:and`. Take the example of images and personal images. A personal image could be described using logical constraint components as described in Section 5.3.7.

[18]https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/If-Modified-Since
[19]https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/If-Unmodified-Since

```
ex:PictureShape
  a sh:NodeShape .

ex:WhatMakesPicturePersonalShape
  a sh:NodeShape .

ex:PersonalPictureShape
  a sh:NodeShape ;
  sh:and (
   ex:pictureShape
   ex:WhatMakesPicturePersonalShape
  ) .
```

Listing 22: SHACL description using logical constrained components

As an example, for the case described in Listing 22, a query engine could cache the evaluation result of `ex:Picture`. Optimizations in the descriptions like this could likely be automated.

Filter Collections on Store Condition

The complexity of filtering the list of eligible collections could be significant. We do, however, expect that this list will be small. In case the `state required` condition is used, a whole SPARQL query needs to be executed to check the state. We will thus disregard that case here. The worst-case performance is listed below:

- Always Stored: Constant
- Prefer Other: linear search in a list of eligible collections.
- Prefer Most Specific: linear scan trough eligible collections and distance function dependent cost for each collection. The distance could be cached.
- Only stored when not redundant: linear scan through collections in case no collection is clear-cut
- Never: constant

Compute Named Node

For each collection that will store the resource, we now have to compute the named node. In the case of URI templates with regexes, this cost negligible.

Create Resources

The Solid Specification requires updates to happen using N3Patch, this means that each created resource requires its own HTTP request.

Interestingly, some implementations of a solid server, like the Community Solid Server[20] also accept SPARQL update queries.

Conclusion Resource Creation

We now know that the resource creation takes 2 HTTP requests: reading SGV and creating the resource. This is compliant with our hypothesis.

6.2.2 Update Resource, No Move Required

This section theoretically analyses the cost of updating a resource when the resource needs not be moved. A general update flow can be found in Chapter 5.2. An example update query is Listing 30 in the empirical evaluation.

Fetch the Description and the Resource

Like with creating a resource, we need to fetch the SGV, costing us one HTTP request. Additionally, we need to fetch the current state of the resource, costing us one additional HTTP request. Luckily, these requests can be done in parallel, minimizing delay.

Construct the result

We then construct the result using the default query engine. The cost of this construction depends on the query engine and is not covered in the work. For the Comunica query engine, the cost of local construction is low when the query engine can be reused.

Check the Update Condition

We know the canonical collection this RDF resource is stored in because the prefix of the collection and the resource named node matches. We then have to check the update condition the resource matches and check if, and how, we can update. In most cases, this is a fairly simple process.

In the case of Keep Distance, an update that stretches the shape description too might cause many updates. It's important to note though that the Amortized Computational Complexity [40] would still make this a constant operation.

Commit Changes

In case no move is required, a single N3Patch request should suffice to update the resource. However, because the primary focus of Comunica is in querying, their update implementation seems to require two, non-parallel HTTP requests[21].

[20]https://communitysolidserver.github.io/CommunitySolidServer/7.x/usage/example-requests/#patch-modifying-resources

[21]As seen in the discussion on https://github.com/comunica/comunica/pull/1326. Will be fixed in the next major release: https://github.com/comunica/comunica/pull/1331

Conclusion Resource Update, No Move

We can conclude that the cost of an RDF resource update is three in the case of an update that only deletes or adds triples, and four in the case of an update that both deletes and updates. It should, however, be possible to do it using only three HTTP requests. Which is valid with our hypothesis, since a non-SGV query engine would require either two or three requests. One to get the original resource, and one or two to update.

6.2.3 Update Resource, Move Required

In the previous section, we assume the update condition concludes no move is required. This section describes the cost when a move is required. In this case, we delete the original RDF resource and follow the steps of Section 6.2.1, disregarding the SGV fetch step.

Assuming we use N3Patch, and the RDF resource is moved to by a different HTTP resource, the required number of requests will be four. One for getting the SGV description (cacheable), one for getting the deleting the resource, one for deleting the original resource, and one for creating the updated resource. When a resource would be moved without SGV, a client would also require three requests, one to get the resource, one to delete the old, and one to insert the new resource. As a result, our hypothesis is still valid.

6.2.4 Conclusion theoretical evaluation

We can thus conclude that our hypothesis about the number of HTTP requests is valid. An SGV client requires at most double the number of HTTP requests a non SGV client requires.

6.3 Empirical Evaluation

After a theoretical evaluation, we also evaluate the implementation in an empirical way. We perform time benchmarks for different queries, all following our use case. The goal of this evaluation is to convince the reader the cost of SGV on query execution is manageable. The hypothesis (Chapter 1.4) is that the execution time for the same query is at most four times as high when using the SGV enabled query engine.

The empirical evaluation is performed using SolidBench[22], which uses the same data as our use case, namely, the Social Network Benchmark [11]. After the generation of our test data, we use SolidBench to host the data locally. Under the hood, SolidBench will use the Community Solid Server [41] to expose the resources.

In our evaluation, we will focus on the RDF resource of a post. Listing 23 provides the ShEx shape of a post. Different pods will have different ways of storing these posts, called fragmentation strategies. We will use four fragmentation strategies in our evaluation:

1. Posts are grouped in files based on the creation date. Within that file, they have a fragment based on the ID. (See Listing 24)
2. Posts are grouped in files based on the location. Within that file, they have a fragment based on the ID. (See Listing 25)
3. All posts are stored in one file. Within that file, they have a fragment based on the ID. (See Listing 26)
4. Each posed is stored in their own file based on the ID. (See Listing 27)

In hindsight, the scope of SolidBench was too big as it creates 1528 pods, but we will only query 4 of them. Each SGV file contains approximately 33 triples. For the writing data use case, the accessed data files are either empty, or in the case of the “one file” fragmentation strategy contain 2947 triples. When updating, we first prepare the file with the insertion of a single post, adding another 9 triples to each data file.

```
prefix ex: <http://example.org/>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ldbc: <http://localhost:3000/www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/>
ex:PostShape {
  ldbc:browserUsed xsd:string ;
  ldbc:creationDate xsd:dateTime ;
  ldbc:hasCreator @ex:PersonShape ;
  ldbc:id xsd:long ;
  ldbc:isLocatedIn @dbo:PlaceShape ;
  ldbc:locationIP xsd:string ;
  ldbc:content xsd:string ? ;
  ldbc:length xsd:int ? ;
  rdfs:seeAlso @sh:IRI * ;
  ldbc:language xsd:string ? ;
}
```

Listing 23: ShEx description of a post

[22]https://github.com/SolidBench/SolidBench.js

```
<posts/> a sgv:canonical-collection ;
  sgv:group-strategy
    [
      a sgv:group-strategty-uri-template ;
      sgv:uri-template
        '{ldbc:creationDate:10}#{ldbc:id}' ;
    ] .
```

Listing 24: Group strategy - by creation date

```
<posts/> a sgv:canonical-collection ;
  sgv:group-strategy [
      a sgv:group-strategty-uri-template ;
      sgv:uri-template
        '{ldbc:isLocatedIn}#{ldbc:id}' ;
      sgv:regex-match '([^/]+)#([^#]+)$' ;
      sgv:regex-replace '$1/$2' ;
    ] .
```

Listing 25: Group strategy - by locations

```
<posts> a sgv:canonical-collection ;
  sgv:group-strategy
    [
      a sgv:group-strategty-uri-template ;
      sgv:uri-template
        '#{ldbc:id}' ;
    ] .
```

Listing 26: Group strategy - one file

```
<posts/> a sgv:canonical-collection ;
  sgv:group-strategy
    [
      a sgv:group-strategty-uri-template ;
      sgv:uri-template '{ldbc:id}' ;
    ] .
```

Listing 27: Group strategy - own file

6.3.1 Test Hardware Specification

For completeness's sake, we briefly describe the system used in the benchmarking. The benchmarks are performed on a Dynabook Inc. Satallite Pro A50EC with 16 GiB memory, an Intel® Core™ i5-8250U x 8 processor and an Intel® Graphic UHD Graphics 620 (KBL GT2). The installed operating system is a Fedora Workstation 39 (64-bit), and firmware version 2.70. It should further be noted that both the query engine and SolidBench run on this machine. As a result, our benchmark does not truly capture the large delays an HTTP request causes over a real network.

6.3.2 Choke Point Queries

We evaluate the vocabulary using multiple queries, each query testing a specific choke point. The choke points we will be testing are:

1. **Create new resource**: Listing 28
2. **Update resource, no move**: Listing 30, Listing 32, Listing 35, Listing 37
3. **Update resource, move**: Listing 33
4. **Illegal update resource**: Listing 31, Listing 36
5. **Delete resource**: Listing 29, Listing 34

The queries should cover all different queries from the update SPARQL specification (see Section 2.2.4). Because we want to cover all types of queries, some choke points are represented by more than one query.

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix card: <http://localhost:3000/pods/0000000000000000
096/profile/card#>
prefix tag: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/tag/>
PREFIX resource: <http://localhost:3000/dbpedia.org/
resource/>

INSERT DATA {
 <> a ns1:Post ;
   ns1:browserUsed "Chrome" ;
   ns1:content
     "I want to eat an apple." ;
   ns1:creationDate
"2024-05-08T23:23:56.83Z"^^xsd:dateTime ;
   ns1:id "416608218494388"^^xsd:long ;
   ns1:hasCreator card:me ;
   ns1:hasTag tag:Alanis_Morissette, tag:Austria ;
   ns1:isLocatedIn resource:China ;
   ns1:locationIP "1.83.28.23" .
}
```

Listing 28: insert data - complete post

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix card: <http://localhost:3000/pods/0000000000000000
096/profile/card#>
prefix tag: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/tag/>
prefix resource: <http://localhost:3000/dbpedia.org/
resource/>
prefix res: <http://localhost:3000/pods/00000000000000000
96/posts/2024-05-08#>

DELETE DATA {
   res:416608218494388
       a ns1:Post ;
       ns1:browserUsed "Chrome" ;
       ns1:content
         "I want to eat an apple." ;
       ns1:creationDate
"2024-05-08T23:23:56.83Z"^^xsd:dateTime ;
       ns1:id "416608218494388"^^xsd:long ;
       ns1:hasCreator card:me ;
       ns1:hasTag tag:Alanis_Morissette, tag:Austria ;
       ns1:isLocatedIn resource:China ;
       ns1:locationIP "1.83.28.23" .
}
```

Listing 29: delete data - complete post

```
prefix tag: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/tag/>
prefix res: <http://localhost:3000/pods/00000000000000000
96/posts/2024-05-08#>

INSERT {
   res:416608218494388 ?p tag:Cheese
} where {
   res:416608218494388 ?p tag:Austria
}
```

Listing 30: insert where - insert tag where tag

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix res: <http://localhost:3000/pods/00000000000000000
96/posts/2024-05-08#>

INSERT DATA {
   res:416608218494388 ns1:id
"416608218494389"^^xsd:long ; .
}
```

Listing 31: insert data - an id (illegal)

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix tag: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/tag/>
prefix res: <http://localhost:3000/pods/000000000000000000
96/posts/2024-05-08#>

INSERT DATA {
    res:416608218494388  ns1:hasTag tag:Mountain .
}
```

Listing 32: insert data - additional tag

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix res: <http://localhost:3000/pods/000000000000000000
96/posts/2024-05-08#>

DELETE {
    ?id ns1:id "416608218494388"^^xsd:long .
} INSERT {
     ?id ns1:id "416608218494389"^^xsd:long .
} where {
    BIND(res:416608218494388 as ?id)
}
```

Listing 33: delete insert - replace id

```
prefix res: <http://localhost:3000/pods/000000000000000000
96/posts/2024-05-08#>

DELETE WHERE {
  res:416608218494388 ?p ?o
}
```

Listing 34: delete where - complete post

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix res: <http://localhost:3000/pods/000000000000000000
96/posts/2024-05-08#>

DELETE {
    res:416608218494388 ns1:hasTag ?x
} where {
    res:416608218494388 ns1:hasTag ?x
}
```

Listing 35: delete - tags

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix res: <http://localhost:3000/pods/000000000000000000
96/posts/2024-05-08#>

DELETE DATA {
    res:416608218494388 ns1:id
"416608218494388"^^xsd:long ; .
}
```

Listing 36: delete data - id (illegal)

```
prefix ns1: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/vocabulary/>
prefix tag: <http://localhost:3000/www.ldbc.eu/ldbc_
socialnet/1.0/tag/>
prefix res: <http://localhost:3000/pods/000000000000000000
96/posts/2024-05-08#>

DELETE DATA {
    res:416608218494388 ns1:hasTag tag:Mountain .
}
```

Listing 37: delete data - a tag

6.3.3 Choke Point: Create New Resource

The creation of a new resource is a type of query where we know we should find the collections the resource belongs to and store it there. The non-SGV variant of query Listing 28 simply replaces the base URI with the named node we decided for the resource. The execution time results are given in Table 3. As expected, the SGV engine is always slower. It is, however, still within the expected range. The ratios are in respective order: 0.629 ; 0.857 ; 0.769 ; and 0.657 . Interestingly, we see that the execution time varies more between fragmentation strategies than it does between using SGV or not.

Task	ops/sec	Average Time (ms)	Margin
insert data complete by creation date: SGV	22	44582.068	±1.73%
insert data complete by creation date: RAW	35	27899.513	±2.07%
insert data complete all in one file: SGV	6	149415.739	±2.98%
insert data complete all in one file: RAW	7	134361.192	±8.66%
insert data complete own file: SGV	10	91851.395	±2.56%
insert data complete own file: RAW	13	76672.217	±3.07%
insert data complete by creation location: SGV	23	43005.366	±2.20%
insert data complete by creation location: RAW	35	28003.949	±2.53%

Table 3: Average execution time of insert data complete query (Listing 28) over 100 runs

6.3.4 Choke Point: Update Resource, No Move

A different kind of choke point is simply a query that results in the same behaviour under SGV as it does without SGV. We evaluate a set of queries that simply modify the resource without moving it in any way.

Table 4, Table 5, Table 6, and Table 7 show the execution time of respectively query Listing 30, Listing 32, Listing 35, and Listing 37. Each of those execution times is lower than those in the previous section. The ratios of SGV-operations over non-SGV-operations per second are:

- For Table 4: 0.467 ; 0.5 ; 0.467 ; and 0.467
- For Table 5: 0.333 ; 0.375 ; 0.333 ; and 0.333
- For Table 6: 0.467 ; 0.5 ; 0.467 ; and 0.467
- For Table 7: 0.333 ; 0.375 ; 0.333 ; and 0.333

Although these ratios are still better than the hypothesized 0.25, they are significantly worse than the previous section. That's to be expected because the SGV enabled a query engine has to perform more steps now. Even tough it is worse across the board, we still see that the fragmentation strategy plays a roll.

Task	**ops/sec**	**Average Time (ms)**	**Margin**
insert where tag by creation date: SGV	7	130209.519	±0.33%
insert where tag by creation date: RAW	15	64915.564	±1.29%
insert where tag all in one file: SGV	4	226277.836	±2.64%
insert where tag all in one file: RAW	8	122255.749	±2.96%
insert where tag own file: SGV	7	129497.887	±0.75%
insert where tag own file: RAW	15	65628.874	±1.45%
insert where tag by creation location: SGV	7	130342.417	±0.36%
insert where tag by creation location: RAW	15	64542.496	±1.33%

Table 4: Average execution time of insert where tag query (Listing 30) over 100 runs

Task	**ops/sec**	**Average Time (ms)**	**Margin**
insert data tag by creation date: SGV	5	178149.073	±0.70%
insert data tag by creation date: RAW	15	63309.807	±0.45%
insert data tag all in one file: SGV	3	289314.253	±0.77%
insert data tag all in one file: RAW	8	123444.536	±3.48%
insert data tag own file: SGV	5	178175.593	±0.28%
insert data tag own file: RAW	15	63045.682	±0.41%
insert data tag by creation location: SGV	5	178955.959	±0.54%
insert data tag by creation location: RAW	15	63363.918	±0.51%

Table 5: Average execution time of insert data tag query (Listing 32) over 100 runs

6.3.5 Choke Point: Update resource: Move

Task	**ops/sec**	**Average Time (ms)**	**Margin**
delete where tags by creation date: SGV	7	128534.614	±0.59%
delete where tags by creation date: RAW	15	64838.515	±1.26%
delete where tags all in one file: SGV	4	217007.521	±1.88%
delete where tags all in one file: RAW	8	115368.001	±2.43%
delete where tags own file: SGV	7	130018.987	±0.49%
delete where tags own file: RAW	15	64744.311	±1.37%
delete where tags by creation location: SGV	7	130569.365	±0.94%
delete where tags by creation location: RAW	15	64453.229	±1.25%

Table 6: Average execution time of delete where tags query (Listing 35) over 100 runs

Task	ops/sec	Average Time (ms)	Margin
delete data tag by creation date: SGV	5	174155.254	±0.71%
delete data tag by creation date: RAW	15	63248.866	±0.48%
delete data tag all in one file: SGV	3	292456.437	±1.42%
delete data tag all in one file: RAW	8	120948.758	±3.02%
delete data tag own file: SGV	5	176555.335	±0.61%
delete data tag own file: RAW	15	63792.069	±0.83%
delete data tag by creation location: SGV	5	175975.652	±0.31%
delete data tag by creation location: RAW	15	63362.320	±0.43%

Table 7: Average execution time of delete data tag query (Listing 37) over 100 runs

This choke point is a vital one to defend SGV as it is impossible to write a SPARQL query that would show the same behaviour as the SGV move. SPARQL is unable to select the CBD. Since a move moves the whole CBD to a potentially new HTTP document, and SPARQL is unable to select the CBD, we required a different approach. We compare the SGV query engine to the execution of two queries in parallel, one delete data query and one insert data query.

Table 8 shows the execution times of query Listing 33. The ratios of operations are: 0.636 ; 0.5 ; 0.417 ; and 0.583 We can thus conclude that our hypothesis is still valid. Again, we highlight the difference between fragmentation strategies. Clearly, the delay experienced from loading the large file in the case all posts are stored in the same file is significant.

Task	ops/sec	Average Time (ms)	Margin
delete insert id by creation date: SGV	7	141940.530	±1.28%
delete insert id by creation date: RAW	11	87113.119	±0.75%
delete insert id all in one file: SGV	2	343690.220	±1.70%
delete insert id all in one file: RAW	4	208930.211	±2.04%
delete insert id own file: SGV	5	177991.908	±0.58%
delete insert id own file: RAW	12	80729.940	±1.06%
delete insert id by creation location: SGV	7	133052.120	±0.60%
delete insert id by creation location: RAW	12	81066.196	±1.15%

Table 8: Average execution time of delete insert ID query (Listing 33) over 100 runs

Task	ops/sec	Average Time (ms)	Margin
insert data id by creation date: SGV	12	80325.290	±0.66%
insert data id by creation date: RAW	15	63787.431	±0.58%
insert data id all in one file: SGV	5	184135.528	±0.71%
insert data id all in one file: RAW	8	122780.884	±2.99%
insert data id own file: SGV	12	79745.654	±0.62%
insert data id own file: RAW	15	63785.574	±0.58%
insert data id by creation location: SGV	12	80393.959	±0.86%
insert data id by creation location: RAW	15	63705.068	±0.66%

Table 9: Average execution time of insert data ID query (Listing 31) over 100 runs

6.3.6 Choke Point: Illegal Update Resource

In this choke point, we evaluate queries that are rejected by an SGV engine. The behaviour is again different from a non-SGV-aware engine. The resulting resources of a normal engine are considered wrong. The execution time ratios are good in this case because the SGV engine need not apply the changes. It can therefore terminate execution early. We have two queries, namely Listing 31 and Listing 36, their execution times can be found in respectively Table 9 and Table 10. The ratios are:

- for Table 9: 0.8 ; 0.625 ; 0.8 ; and 0.8
- for Table 10: 0.8 ; 0.625 ; 0.8 ; and 0.8

These results confirm our hypotheses.

Task	ops/sec	Average Time (ms)	Margin
Delete data id by creation date: SGV	12	80141.498	±0.80%
Delete data id by creation date: RAW	15	63500.094	±0.43%
Delete data id all in one file: SGV	5	185053.673	±0.73%
Delete data id all in one file: RAW	8	121587.804	±2.94%
Delete data id own file: SGV	12	79697.409	±0.60%
Delete data id own file: RAW	15	63715.705	±0.92%
Delete data id by creation location: SGV	12	79515.830	±0.51%
Delete data id by creation location: RAW	15	63687.341	±0.77%

Table 10: Average execution time of delete data ID query (Listing 36) over 100 runs

Task	ops/sec	Average Time (ms)	Margin
delete data complete by creation date: SGV	5	177951.642	±0.53%
delete data complete by creation date: RAW	14	67166.339	±0.55%
delete data complete all in one file: SGV	3	298814.885	±0.94%
delete data complete all in one file: RAW	8	119370.476	±2.83%
delete data complete own file: SGV	5	176949.144	±0.33%
delete data complete own file: RAW	14	66844.005	±0.59%
delete data complete by creation location: SGV	5	178498.626	±0.30%
delete data complete by creation location: RAW	15	65565.496	±0.76%

Table 11: Average execution time of delete data complete query (Listing 29) over 100 runs

6.3.7 Choke Point: Delete Resource

Queries under this choke point have the same behaviour for both an SGV-enabled engine and an engine that is not SGV-enabled. Table 11 and Table 12 show the execution time for respectively query Listing 29 and Listing 34. The ratio of operations between SGV and raw are:

- Table 11: 0.357 ; 0.375 ; 0.357 ; and 0.333
- Table 12: 0.467 ; 0.5 ; 0.5 ; and 0.467

These ratios confirm our hypothesis.

6.3.8 Conclusion

In conclusion, our hypothesis holds when we compare the execution time and HTTP request count of an SGV query engine to a non-SGV engine that executes the same operations that the SGV engine would take. Unfortunately, when a move of the CBD of a resource is required, a developer cannot use a SPARQL query engine, since SPARQL is not expressive enough to describe the CBD. In case such behaviour is desired, a manual interaction with the interface is required.

A different approach might be to use the "DESCRIBE" query of SPARQL that is sometimes implemented as the CBD of a resource. However, since this choice is imple-

Task	ops/sec	Average Time (ms)	Margin
delete where complete by creation date: SGV	7	132988.158	±0.46%
delete where complete by creation date: RAW	15	65584.642	±0.40%
delete where complete all in one file: SGV	4	213726.655	±1.36%
delete where complete all in one file: RAW	8	121148.118	±2.93%
delete where complete own file: SGV	7	130769.232	±0.43%
delete where complete own file: RAW	14	69931.699	±3.13%
delete where complete by creation location: SGV	7	130786.473	±0.62%
delete where complete by creation location: RAW	15	66160.665	±1.28%

Table 12: Average execution time of delete where complete query (Listing 34) over 100 runs

mentation-specific, and is not required by the SPARQL spec, using describe to get the CBD is not advised.

7 Future Work

SGV proves it is possible to create automated clients that can decide where to store resources. In the state we present it here, it is not production ready, but it opens the gate to interesting research.

7.1 Source Discovery

In this work, we do not discuss source discovery when updating a resource. In our implementation, we take the classical approach where you need to provide a resource that should be updated. This means we actually still have a data access path dependency! The only access path dependency we solved is the one where a resource is created. When preforming an update, we expect the user to update only one specifically specified resource at a time. This way, they will know the Named Node of the resource, and therefore they know where it is stored.

The problem is that a user does not always know the resource they want to update. Imagine changing your name, you would like to update all HTTP resources that contain your name. How do you know what resources contain your name? In the context of read queries, one might want to preform a link traversal query over their pod. A query engine should be able to do something similar for the case of update queries. When confronted with a query to change your name, it should find all documents containing your name and alter them. Finding these resources can happen using any source discovery technique. When constructing the result, the query engine should keep a source attribution list[23], and update these sources. The construction of a source attribution list is related to the domain of data provenance [42], which is well established research within the semantic web community [43].

7.2 Inter pod updates

SGV is restricted to updating a single pod. Additional research should go into updates that alter multiple pods. Handling multiple pods is complex as many decisions are valid. In the example of two pods, there is already a multitude of use cases, each with different considerations.

[23]Interestingly, this is currently being implemented in Comunica:
https://github.com/comunica/comunica/pull/1325

1. As a pod owner, I want to transfer pictures I have to someone else, so they now own that picture. Note that I am not guaranteed to have write permissions to the other Solid pod.
2. As a pod owner, I want to transfer a token to a pod I do, or do not, have write access to. The token should always exist *exactly once*, meaning there is always one person holding the token, and everyone can see who has it.
3. As a pod owner, I want to insert an additional property to an existing resource in someone else's pod. For example, I transferred a picture and forgot to add a description.
4. As a pod owner, I want to delete a property of an existing resource in someone else's pod.
5. As a pod owner, I want to remove a resource in someone else's pod, so I don't see it anymore. Essentially, I want to change my view of the resource. This could be achieved by using the Subweb Specification [44] and adding a rule that makes me ignore the "virtually" deleted triple.
6. As a pod owner, I want to remove a resource in someone else's pod, so no one can see it. I might want to send a suggestion in a notification collection of the targeted pod.

Besides access control problems and how to circumvent them, we also face the problem of backlinks. We already suggested the use of an sgv:moved-to predicate to prevent links from breaking, but it might be better to discover backlinks and alter them when a resource changes.

7.3 Other Interfaces

This work focusses on LDP interfaces. For such an interface, data is linked to newly created tuples through ldp:contains predicates. When we don't use LDP, or use a different kind of interface, the question we try to answer might shift from "Where do I store this resource?" to "What other resources are linked to this new one, and through what predicates"?

There is merit to investigating different interfacing technologies because LDP is far from perfect. By nature, LDP restricts data consumers [34]. Even more so, much of the complexity of SGV is required because of LDPs nature. Nevertheless, it is unlikely that

LDP completely disappears because the low server complexity makes it very attractive for data providers.

Through SGV, it would also be possible to create multiple interfaces on the same data. You could for example expose the raw data graph of a pod through a SPARQL endpoint. The endpoint could then, for example, only be used by highly authorized users. Another interface could be LDP powered and be constructed based on an SGV description of derived collections.

7.4 Guided queries

Within the research around querying the semantic web, there exists link traversal. Unlike federated querying where you define the sources to query over beforehand, link traversal will discover new sources while executing queries. This comes with various difficulties, like safety issues [38], completeness modelled by completeness guarantees, and execution time. There has been the assumption that we cannot reduce the execution time of a link traversal powered query over the semantic web because of its enormous size. The consensus has thus been that we should just make sure most results are received fast through link prioritization [45], that way we can set a timeout and assume no results would be found after a time. Recent work that uses completeness guarantees and the structured nature of some interfaces has shown that it is possible to speed up queries and be complete to a certain extent [21]. That early work uses type indexes to get structural descriptions, but the complexity could be increased to use shape trees or even SGV. In this extension, SGV can prove to be more valuable than shape trees because it expresses the underlying data flow better. For example, a collection that is derived from another should not be consulted if the canonical containers have already been consulted.

7.5 View Creation and Discovery

The issues related to the document-based nature of the current Solid specification that have been described [34] can be solved by creating derived resources [35]. The work by J. Van Herwegen and R. Verborgh shows that derived resources are a way forward. Given the similarities between their work and SGVs derived resources, we are confident that an implementation is feasible. In their work, they solve the issue of access control granularity.

In our empirical evaluation, we discover that the execution time of our queries heavily relies on our pod structures (Section 6.3.3). We therefore expect that a smart server that knows what optimizations are possible by query engines could have significant execution time benefits. Such a server could create resources dynamically to facilitate query executions. The resources to create could be based on the usage metrics the server has.

7.6 Smart Access Control

Both WAC and ACP don't allow users to create access control rules based on the RDF content itself. Since a long time, efforts exist to change this, one such effort is the Universal Access Control[24] of Thomas Bergwinkl. A motivation for disallowing these kinds of policies could be the rise in user complexity. However, through the resource description formats like Shape Trees and SGV, one could argue that we express what data is in a resource. Therefore, we could extract an access control policy in function of the data based on the policy on the document.

Access control extraction could help create uniform access rules across multiple interfaces of the same pod in a way that feels familiar for users.

7.7 SGV Integration with Existing Structure Ontologies

The vocabulary described in this work has limited interoperability with existing vocabularies. Since SGV exists in the same domain as shape trees [23], it would make sense to adapt/ extend the vocabulary in such a way that it could be easily plugged into an existing shape tree environment. This alternative structure would likely be less expressive.

In the same way, integration with TREE [13] would increase the vocabulary's interoperability. It should, however, be noted that Tree could also be considered “a kind of structured data you can store using LDP and SGV.”

[24]https://www.bergnet.org/people/bergi/files/documents/2014-02-14/index.html#/

7.8 General Update Behaviour

The question we asked ourselves when starting this work was: "how can we make updating solid pods easier?" We ended up creating a base layer that allows query engines to decide how to store resources. That was not the only possible way of making updates easier. In this section, we list a few more possible improvements related to data updates. May it inspire anyone to work on these challenging topics.

7.8.1 CRDTs: The Eventual Consistency Approach

Through Solid, many applications are working on the same data, and each application likely has their own cache in place. As a result, applications working on the same data all have their own local copy of the data, essentially creating a distributed system. It is important that one application does not just undo the work by another application. A CRDT (*Conflict-free Replicated Data Type*) is a data type with the properties that essentially chooses for eventual consistency on the CAP scale [3]. A basic CRDT implementation for Solid[25] already exists, recently created by Noel De Martin, hosting the vocabulary online[26]. That implementation is a nice starting point, but it does not yet contain logical clocks.

7.8.2 ACID Transactions

Massive adaptation is the dream of any technology, but to achieve that, you need to be at least as good as the competition. The largest competitor for data storage is the relational database offering the ACID (*atomicity, consistency, isolation, durability*) properties. Not only do developers expect these properties, many applications are unable to operate without these consistency guarantees. We therefore advocate for research into stronger consistency guarantees in Solid.

Such a claim is not always well received, because of the CAP theorem that states that you can only have two out of {Consistency, Availability, Partition tolerance} [3]. However, the choice is not binary, as later clarified by the writers of the CAP theorem [46]. Not only can the research decide in "how consistent" we want to be, we have a varying partition tolerance variable too! Most distributed database systems replicate data across machines so that when one machine goes down, all data is still accessible,

[25]https://slidr.io/NoelDeMartin/solid-crdts-in-practice#36
[26]https://vocab.noeldemartin.com/crdt/

be it through other nodes in the system. Solid does not have such a replication system in place. When a single pod disconnects from the network, the data on that pod cannot be accessed until the pod connects to the network again. We believe that this opens some space for research on stronger consistency requirements.

The decision of what point in the CAP space we work with need not be done at pod level, but could be done on HTTP resource level. For example, one HTTP resource might support CRDTs essentially choosing for availability over consistency. Another resource might introduce some locking mechanism, choosing consistency over availability.

8 Conclusion

In this work, we presented a vocabulary that allows smart clients to autonomously discover the location a created or updated resource should be stored. The vocabulary also introduces checks on whether a resource can be created or removed. Additionally, we proved that our vocabulary is indeed expressive enough by implementing a smart client that consumes it.

We hypothesized that such a smart client would be a maximum of four times slower and would require a maximum of double the amount of HTTP requests. Through theoretical evaluation, we discovered that the amount of HTTP requests is within those bound. Using empirical evaluation, we also validated that the execution time overhead is within the accepted range. Moreover, we saw that some of SGVs behaviour cannot be modelled using a SPARQL query.

In essence, SGV tries to provide structure to a widely unstructured document store that is LDP. It does this by defining a server-side description of the structure that should be enforced by clients. In reality, clients can still interact with the Solid pod however they want, since the server is not aware that a structure should be followed. This lack of server-side verification is perhaps the biggest shortcoming of this work. That being said, it is entirely possible to extend an existing Solid server with a SGV verification system. The downside at that being that both the client and server need to calculate the proposed location of a resource. Unfortunately, this is a shortcoming of the LDP interface itself, as it chooses for a low-complexity server. This choice often comes at the cost of a complex client side. What's more, since one server interface is used by many clients, it becomes almost impossible to guarantee a system that respects the structure of a permissive interface.

Throughout this work, we frequently talk about "the RDF resource", defining it as the CBD of some named node. This definition is actually rather arbitrary. Another way of defining a resource is by using the shape descriptions. Though the use of shape descriptions, create an RDF resource containing multiple named nodes as subjects. Listing 38 shows a description with multiple named nodes. SGV will then try to define a base for this query and place the resource there. When editing the resource, we need

```
@prefix ex: <http://example.org/> .
<> a ex:Person ;
  ex:address <#myAddress> .

<#myAdress> a ex:Address ;
  ex:street "SesamStreet"@en .
```

Listing 38: A resource consisting of two named nodes as subjects.

to be aware that both named nodes are referable by others. Thus, when we conclude a move is required, we should decide on what named nodes should be moved.

Bibliography

[1] J. Gray and others, “The transaction concept: Virtues and limitations,” in *VLDB*, 1981, pp. 144–154.

[2] M. Bosquet, “Access Control Policy (ACP),” May 2022.

[3] A. Fox and E. A. Brewer, “Harvest, yield, and scalable tolerant systems,” in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, 1999, pp. 174–178.

[4] P. Stickler, “CBD - Concise Bounded Description,” Jun. 2005.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, 2011, pp. 386–400.

[6] P. Colpaert, “Linked Data Event Streams.”

[7] S. Speicher, J. Arwe, and A. Malhotra, “Linked Data Platform 1.0,” Feb. 2015.

[8] D. Wood, M. Lanthaler, and R. Cyganiak, “RDF 1.1 Concepts and Abstract Syntax,” Feb. 2014.

[9] H. Knublauch and D. Kontokostas, “Shapes Constraint Language (SHACL),” Jul. 2017.

[10] T. Baker and E. Prud'hommeaux, “Shape Expressions (ShEx) 2.1 Primer,” Oct. 2019.

[11] O. Erling *et al.*, “The LDBC social network benchmark: Interactive workload,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 619–630.

[12] R. Verborgh *et al.*, “Triple pattern fragments: a low-cost knowledge graph interface for the web,” *Journal of Web Semantics*, vol. 37, pp. 184–206, 2016.

[13] P. Colpaert, “The TREE hypermedia specification,” Apr. 2024.

[14] R. Cyganiak, J. Zhao, M. Hausenblas, and K. Alexander, “Describing Linked Datasets with the VoID Vocabulary,” Mar. 2011.

[15] T. Berners-Lee, H. Story, and S. Capadisli, “Web Access Control,” May 2024.

[16] R. Verborgh, “Re-Decentralizing the Web, For Good This Time,” *Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web*. ACM, pp. 215–230, Sep. 2023. doi: 10.1145/3591366.3591385[27].

[17] E. Mansour *et al.*, “A Demonstration of the Solid platform for Social Web Applications,” in *Proceedings of the 25th International Conference Companion on World Wide Web*, 2016, pp. 223–226.

[18] M. Kleppmann *et al.*, “Bluesky and the AT Protocol: Usable Decentralized Social Media.” 2024.

[19] M. Zignani, S. Gaito, and G. P. Rossi, “Follow the "Mastodon": Structure and Evolution of a Decentralized Online Social Network,” in *Twelfth International AAAI Conference on Web and Social Media*, 2018.

[20] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Decentralized business review*, 2008.

[21] R. Taelman and R. Verborgh, “Link traversal query processing over decentralized environments with structural assumptions,” in *International Semantic Web Conference*, 2023, pp. 3–22.

[22] T. Turdean, J. Zucker, V. Balseiro, S. Capadisli, and T. Berners-Lee, “Type Indexes,” Jun. 2022.

[23] E. Prud'hommeaux and J. Bingham, “Shape Trees Specification,” Dec. 2021.

[24] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[25] T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web: A New Form of Web Content That is Meaningful to Computers Will Unleash a Revolution of New Possibilities,” *ScientificAmerican.com*, p. , 2001.

[26] G. Carothers and A. Seaborne, “RDF 1.1 N-Triples,” Feb. 2014.

[27] G. Carothers and E. Prud'hommeaux, “RDF 1.1 Turtle,” Feb. 2014.

[28] A. Seaborne and S. Harris, “SPARQL 1.1 Query Language,” Mar. 2013.

[29] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, “Comunica: A Modular SPARQL Query Engine for the Web,” in *Proceedings of the 17th International Semantic Web Conference*, Springer International Publishing, Oct. 2018, pp. 239–255. doi: 10.1007/978-3-030-00668-6_15[28].

[30] S. Capadisli, T. Berners-Lee, R. Verborgh, and K. Kjernsmo, “Solid Protocol,” Dec. 2022.

[31] R. Elmasri, “Fundamentals of database systems seventh edition,” 2021.

[32] J. Park and R. Sandhu, “Towards usage control models: beyond traditional access control,” in *Proceedings of the seventh ACM symposium on Access control models and technologies*, 2002, pp. 57–64.

[33] J. Bingham, E. Prud'hommeaux, and elf Pavlik, “Solid Application Interoperability,” Apr. 2024.

[34] R. Dedecker, W. Slabbinck, J. Wright, P. Hochstenbach, P. Colpaert, and R. Verborgh, “What’s in a Pod? A knowledge graph interpretation for the Solid ecosystem,” in *6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs (QuWeDa) at ISWC 2022*, 2022, pp. 81–96.

[35] J. Van Herwegen and R. Verborgh, “Granular Access to Policy-Governed Linked Data via Partial Server-Side Query.”

[36] R. T. Fielding, M. Nottingham, D. Orchard, J. Gregorio, and M. Hadley, “URI Template.” [Online]. Available: https://www.rfc-editor.org/info/rfc6570[29]

[37] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, “Uniform Resource Identifier (URI): Generic Syntax.” [Online]. Available: https://www.rfc-editor.org/info/rfc3986[30]

[38] R. Taelman and R. Verborgh, “A prospective analysis of security vulnerabilities within link traversal-based query processing,” in *6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs (QuWeDa) at ISWC 2022*, 2022, pp. 33–48.

[39] S. Staworko and P. Wieczorek, “Containment of Shape Expression Schemas for RDF,” in *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, in PODS '19. <conf-loc>, <city>Amsterdam</city>, <country>Netherlands</country>, </conf-loc>: Association for Computing Machinery, 2019, pp. 303–319. doi: 10.1145/3294052.3319687[31].

[40] R. E. Tarjan, “Amortized computational complexity,” *SIAM Journal on Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306–318, 1985.

[41] J. Van Herwegen and R. Verborgh, “The Community Solid Server: Supporting Research & Development in an Evolving Ecosystem.”

[42] R. A. Becker and J. M. Chambers, “Auditing of Data Analyses,” *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 4, pp. 747–760, 1988, doi: 10.1137/0909049[32].

[43] S. Sahoo, T. Lebo, and D. McGuinness, “PROV-O: The PROV Ontology,” Apr. 2013.

[44] B. Bogaerts, B. Ketsman, Y. Zeboudj, H. Aamer, R. Taelman, and R. Verborgh, “Link Traversal with Distributed Subweb Specifications,” in *Proceedings of the 5th International Joint Conference on Rules and Reasoning*, S. Moschoyiannis, R. Peñaloza, J. Vanthienen, A. Soylu, and D. Roman, Eds., in Lecture Notes in Computer Science, vol. 12851. Springer, Sep. 2021, pp. 62–79. doi: 10.1007/978-3-030-91167-6_5[33].

[45] O. Hartig and M. T. Özsu, “Walking without a map: Ranking-based traversal for querying linked data,” in *The Semantic Web–ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I 15*, 2016, pp. 305–324.

[46] E. Brewer, “CAP twelve years later: How the "rules" have changed,” *Computer*, vol. 45, no. 2, pp. 23–29, Feb. 2012, doi: 10.1109/MC.2012.37[34].

[27]https://doi.org/10.1145/3591366.3591385
[28]https://doi.org/10.1007/978-3-030-00668-6_15
[29]https://www.rfc-editor.org/info/rfc6570
[30]https://www.rfc-editor.org/info/rfc3986
[31]https://doi.org/10.1145/3294052.3319687
[32]https://doi.org/10.1137/0909049
[33]https://doi.org/10.1007/978-3-030-91167-6_5
[34]https://doi.org/10.1109/MC.2012.37