Decentralized Infrastructure for Versioned Linked Open Data and Scalable Curation Thereof

A DISSERTATION PRESENTED
BY
SINA MAHMOODI
TO
THE DEPARTMENT OF INFORMATIK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE
IN THE SUBJECT OF
COMPUTER SCIENCE

EXAMINERS:
PROF. HOFMANN-APITIUS
PROF. JENS LEHMANN

ADVISORS:
DR. MARC JACOBS
DR. DAMIEN GRAUX

UNIVERSITY OF BONN

DECEMBER 2018

Abstract

The growing web of data warrants better data management strategies. Data silos are single points of failure and they face availability problems which lead to broken links. Furthermore the dynamic nature of some datasets increases the need for a versioning scheme. In this work, we propose a novel architecture for a linked open data infrastructure, built on open decentralized technologies. IPFS, a P2P globally available filesystem, is used for storage and retrieval of data, and the public Ethereum blockchain is used for naming, versioning and storing metadata of datasets. Triples are indexed via a Hexastore, and Triple Pattern Fragments framework is used for retrieval of data. We furthermore explore two mechanisms for maintaining a collection of relevant, high-quality datasets in a distributed manner in which participants are incentivized. The platform is shown to have a low barrier to entry and censorship-resistance. It benefits from the fault-tolerance of its underlying technologies and in most cases is expected to offer higher availability. An analysis in terms of the FAIR principles, showing improved findability, interoperability and accessibility for datasets published on the infrastructure, is further provided.

Declaration of Authorship

I, Sina Mahmoodi, declare that this thesis, titled "Decentralized infrastructure for versioned linked open data and scalable curation thereof", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work. I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Contents

List of Figures

List of Tables

Listings

1

Introduction

Over the last decade, as more and more linked data in the form of RDF [1] triples were published, a set of data management practices [2] were proposed and adopted which aimed to improve integration and reuse among datasets, forming the web of data, which can be seen as a global namespace connecting individual graphs and statements. The linked data principles [3] recommend naming things by way of HTTP-based URIs, which are dereferenceable, i.e.

return data in form of data dump or a SPARQL [4] endpoint, and include links to other statements. A community effort [5] focuses on publishing datasets under open licenses [6] and interlinking them, which has resulted in the LODCloud [7]. Semantic Sitemaps [8] and voID [9] provide means for specifying metadata, e.g. provenance, license and versioning information, for a dataset. Recently, a wide range of stakeholders came together and proposed the FAIR guiding principles [10] for scholarly data, which comprise of being Findable, Accessible, Interoperable and Reusable.

From a logical point of view, linked data is inherently decentralized. However, from a physical point of view, the actual data reside on data silos which suffer from low availability [11, 12], leading to broken links [13]. Low availability could manifest itself in several forms, such as unreliable service, service going completely offline e.g. due to lack of funding, or timeouts due to high demand and expensive queries putting pressure on public SPARQL endpoints [14]. Data publishers are therefore required to undergo the costs of maintaining a server and ensuring adherence to good data management practices, in turn leading to entry barriers. Shortage of reliable live queryable knowledge graphs makes it difficult for developers to build applications which consume linked data. Furthermore, when considering dynamic datasets [12], a lack of robust versioning scheme can lead to inconsistencies when an external linked dataset is modified. But versioning datasets using HTTP has so far proven difficult [15]. Another implication of the unprecedented volume of data being published in web of data is the varying quality of datasets, which ranges from extensively curated to ones that are of relatively lower quality. Expert quality assessment [16] and curation produces the best result, but in large scale incurs high costs in terms of expert time and labor. A promising affordable approach is combining expert and crowdsourced assessment [17].

Given the recent progress in decentralized technologies, specifically globally available P2P filesystems and public blockchains, and in particular IPFS [18] and Ethereum [19], in this work we explore an alternative architecture for linked data infrastructure.

Utilizing IPFS for storage and retrieval of data offers the following benefits over a data silo:

Authenticated, immutable addressing of data IPFS objects are addressed by their cryptographic hash, therefore the authenticity of data can be verified against the address upon retrieval. Furthermore, any modification of the underlying data results in a completely new address, providing a basis for a versioning scheme.

No single point of failure Data objects can be retrieved as long as there exists at least one node with a replica. Nodes have the choice to ”pin” data objects they interact with, thereby increasing replication for popular datasets.

Incentivizing provable replication Building on IPFS, Filecoin [20] incentivizes nodes to provably store distinct replicas of data objects.

Low barrier to entry Maintaining a highly available server would not be necessarily, but is beneficial to store at least one replica for new datasets.

Built-in data caching Being content-addressed, IPFS objects can be cached long-term by data consumers and used even when offline.

The public Ethereum blockchain can be seen as a decentralized smart contract platform with turing-complete expressiveness, based on a P2P network and a distributed consensus algorithm for reaching agreement on a shared state. Smart contracts, abstractly, are persistent

state machines, with pre-defined procedures for state transition. Key features of Ethereum relevant to the context of this work include fair access, data immutability and integrity [21], and strong data availability guarantees (albeit at a relatively high cost). Digital signatures and public-key cryptography are employed for authorization of pseudo-anonymous accounts. Ethereum further enables trustless large scale coordination of participants around a set of goals, by relying on cryptographic proof of properties of past behaviour and economically incentivizing desired properties in future actions [22].

The contributions of this work include a novel architecture for a decentralized linked open data infrastructure, based on IPFS and the public Ethereum blockchain. The design includes an indexing scheme suitable for linked data, and a mechanism for retrieval of data by performing triple pattern or SPARQL queries. It further outlines how smart contracts can be employed to provide a persistent identifier for data objects stored on IPFS, to describe and version datasets, to control write access and to ensure source of provenance. A prototype of the aforementioned architecture has been implemented, and is available under an open license at `https://github.com/sina/open-knowledge`. The prototype includes a Javascript SDK, which facilitates programmatical interaction with the infrastructure, e.g. to publish or query datasets. On this foundation, and to further explore crowdsourcing data curation in scale, we consider employing two mechanisms, first proposed by Ethereum community members [23, 24], which facilitate distributed, trustless, incentivized consensus on a curated list of datasets. The mechanisms are agnostic to the domain and the actual quality metrics.

The rest of this dissertation proceeds as follows: We discuss related work in chapter 2. To keep the work self-contained, relevant aspects of IPFS and Ethereum protocols are highlighted in chapter 3. Chapter 4 introduces the proposed architecture, and describes how such

an architecture can be implemented. Mechanisms for scalable curation of data are outlined in chapter 5. We then qualitatively discuss the characteristics of the platform, the degree of FAIRness for datasets published on it, and consider attack vectors for one of the curation mechanisms in chapter 6. Chapter 7 concludes the work, and sets forth ideas for future work.

2

Related Work

There has been extensive research on centralized RDF data storage and retrieval. Triple stores such as Jena2 [25] and Sesame [26] fall into this category. A survey of such storage and query processing schemes has been done by Faye et al. [27], in which triple stores are categorized based on multiple factors. These factors include native vs non-native and in-memory vs disk-based storage solutions. Non-native solutions for example are triple stores that use an exist-

ing data store, such as relational databases. Hexastore [28] stores six indices, enabling efficient lookup of triple patterns for each parts of the triple, including subject, predicate and object. This gain in performance comes at a cost in storage. When it comes to querying data from remote servers, Verborgh et al. argue that there's a spectrum between data dumps and SPARQL endpoints, and that there's a trade-off along the spectrum between factors including performance, cost, cache reuse, bandwidth, etc. for servers and clients. They propose Linked Data Fragments [14] which lies somewhere in the middle of the spectrum. In this design, clients turn a SPARQL query into a series of triple pattern requests that servers respond to, lowering load on servers, decreasing bandwidth, etc. Centralized data repositories can process queries very efficiently, but they are single points of failure and they have limited scalability and availability. In this work we adopt core ideas from Hexastore and LDF and apply them to the P2P network setting. Similarly to Hexastore, triples are indexed six times, and stored as a Merkle DAG on IPFS. Furthermore We adopt the Triple Pattern Fragments to deconstruct full SPARQL queries into triple patterns, and fetch the relevant triples from IPFS. The whole aforementioned process is performed on clients.

Content distribution over P2P networks has been an area of active research during the last two decades. Motivations over the client-server architecture include scalability, fault-tolerance, availability, self-organization and symmetry of nodes [29]. Androutsellis-Theotokis et al. classify P2P technologies [30] in the context of content distribution into applications and infrastructure. P2P applications themselves are classified into file exchange applications, such as Napster [31], Gnutella [32] and Kazaa [33], which facilitate one-off file exchange between peers, and content publishing and storage applications, such as Publius [34], Freenet [35] and Oceanstore [36], which are distributed storage schemes in which users can store,

publish and distribute content securely and persistently. Technologies targeted for routing between peers and locating content have been classified under P2P infrastructure, and include Chord [37], CAN [38], Pastry [39] and Kademlia [40]. P2P networks also differ in their degree of centralization. Some, like Napster, rely on a central server which holds metadata crucial for routing and locating content, limiting scalability, fault-tolerance and censorship-resistance, but offering efficient lookups. On the other hand, networks like Gnutella are fully decentralized, i.e. every participating node is equal in terms of capabilities, often offer better scalability, availability and robustness, at the cost of efficiency. Lua et al. review overlay network structures, comparing *structured* and *unstructured* networks [41]. Peers in unstructured networks like Gnutella are connected mostly randomly and they query content by flooding, random walk, etc. Whereas peer neighbours in structured networks are more deterministic and tend to form a well-defined shape. Structured networks employ Distributed Hash Tables (DHTs) and place routing metadata in particular points in the network to achieve more efficient lookups, but by default only support exact queries.

The infrastructure proposed in this work is built on top of IPFS [18]. According to the aforementioned taxonomy, IPFS can be classified as a fully decentralized, structured network for content publishing and storage (as opposed to simple file exchange). For routing, it uses a DHST based on S/Kademlia [42] and Coral [43]. It further uses BitSwap, a block exchange protocol, similar to BitTorrent [44], in which peers roughly follows a tit-for-tat exchange strategy. On top of these, IPFS uses a generalization of the Git data structure, to model objects and links between them via cryptographic hashes, forming a Merkle DAG.

Unstructured P2P networks have been employed in protocols such as Bibster [45] and [46] to store RDF data and process queries. They use semantic similarity measures to form

semi-localized clusters and to propagate queries to peers who are most likely to contain relevant data for. These protocols offer higher fault tolerance, but limited guarantees for retrieving query results even the underlying data exists in the network due to their propagation mechanisms.

Filiali et al. has performed a comprehensive survey [47] of RDF storage and retrieval over structured P2P networks. The underlying network topologies are categorized as ring-based, cube-based, tree-based and generic DHT-based. Focusing on data indexing and query processing as the main challenges, a few common patterns have been observed. To index the triples, most protocols rely on variants of hash-indexing, e.g. RDFPeers [48], or semantic indexing, e.g. GridVine [49]. As DHTs are only suitable for exact queries by default, often additional mechanisms need to be devised to facilitate richer semantic queries. Two general strategies have been observed by Filiali et al., either retrieving all relevant triples from other peers and evaluating the result of the query locally, or propagating the query and partial results through the network, as in QC and SBV [50]. Unlike the aforementioned protocols, in this work we don't design a custom P2P network specifically built for RDF data storage, but use the live global IPFS filesystem, which is simultaneously being used for other purposes. Triples are indexed as a Hexastore. To process queries, all relevant triples are fetched, and result is evaluated using the Triple Pattern Fragments [14] framework. The proposed data storage solution further versions datasets and ensures data integrity upon retrieval.

Sicilia et al. [51] explore publishing datasets on IPFS, either by storing the whole graph as a single object or by storing each dereferenceable entity as an object. Furthermore they propose using IPNS to refer to the most recent version of a dataset. In this work, datasets are indexed in the form of a Hexastore, which allows for efficient retrieval of data, and versioning

is handled by a smart contract on Ethereum.

English et al. [52] explore both utilizing public blockchains for the semantic web, by improving on the current URI schemes, and storing values on the Bitcoin network, and creating ontologies for representing blockchain concepts. We share the idea that blockchains and web of data are complementary and can benefit from one another, and use the Ethereum blockchain to store metadata, and perform curation for knowledge graphs.

To determine ”fitness for use” of a dataset, assessment of its quality is required. Quality is however a broad term and it can encompass a variety of metrics in different contexts. Zaveri et al. has performed a systematic review [16] of quality assessment approaches in the literature, arriving 18 quality dimensions and 69 metrics, which can be used in an expert manual curation. Sieve [53], a module for the Linked Data Integration Framework [54], can be used in an integration pipeline to keep, discard on transform values based on specification of quality assessment metrics as specified by the user. Another promising direction is detecting and repairing data quality issues automatically, by adapting ideas from test-driven software engineering, to data engineering, which has been shown [55] to efficiently reveal many issues in existing datasets. When expert manual curation using quality metrics is prohibitively costly, and automatic detection doesn't cover all of the issues, crowdsourcing curation has been shown [17] to be an affordable way of enhancing quality, complementing expert curation. In this work, two mechanisms that rely on a blockchain as a trustless mediator are considered for community-based, distributed and incentivized curation of data in large scale, the result of which is collections of datasets that are relevant for a context, or that satisfy certain quality metrics.

3

Background

3.1 IPFS

IPFS[18] is a peer-to-peer protocol for content-addressable storage and retrieval of data. Detailed workings of IPFS is out of the scope of this work, however the relevant aspects of the distributed file system will be highlighted here.

3.1.1 Network Topology

IPFS is a peer-to-peer network, with no difference between the participating nodes. It utilizes routing mechanisms (such as a DHT) to keep track of data storage, and block exchange mechanisms to facilitate the transfer of data. Every node stores IPFS objects in their local storage.

3.1.2 Replication

These objects could be published by the node, or retrieved from other nodes and replicated locally. It's important to note that every node only stores objects which they care about, and not an arbitrary subset of the whole data set.

3.1.3 Content Addressing

Objects in IPFS are comprised of immutable content-addressed data structures (such as files), that are connected with links, forming a Merkle DAG (directed acyclic graph). Addressing is done using cryptographic hashes. Most often the hash function used is SHA-256, however, as the protocol is designed using self-describing identifiers, the protocol is agnostic to the actual function. The links that connect the objects, are themselves hashes of the target object, which is stored in the source object.

The aforementioned model lends several important characteristics to the IPFS protocol. Content can be identified uniquely by its hash, and after retrieval, the integrity of it can be verified against the hash that was used to address it. Furthermore, even the smallest change in an addressed content, results in a completely new object, with a new hash. Therefore, the original version of the content remains intact, giving way to object permanency.

IPFS, however, does not guarantee persistence, only permanence. A piece of content can always be referred by its hash, but it doesn't necessarily exist in the nodes of the network at all times.

3.2 IPLD

IPLD* is a data model that aims to provide a unified address space for hash-linked data structures, such as IPFS objects, git objects, Ethereum transaction data, etc., which would allow traversing data regardless of the exact underlying protocol.

The benefits of such a data model include protocol-independent resolution and cross-protocol integration, upgradability, self-descriptive data models that map to a deterministic wire encoding, backward-compatibility and format-independence.

3.2.1 CID

A key aspect of IPLD, is a self-describing content-addressed identifier format, called CID†.

In short, a CID describes an address along with its base, version of the CID format, format of the data being addressed, hash function, hash size and finally the hash (address). This allows CID to address objects from various protocols.

As an example, the CID `z43AaGF23fmvRnDP56Ub9WcJCfzSfqtmzNCCvmz5eudT8dtdCDS`, describes the Ethereum block hash `0x8a8e84c797...` in v1 format of CID, has `eth-block` as data format and `keccak256` as hash function.

*https://github.com/ipld/specs
†https://github.com/ipld/cid

3.2.2 Data Model

IPLD defines merkle-dag, merkle-links and merkle-paths among other things. Merkle-dag is as we saw before, a graph, the edges of which are merkle-links. A merkle-path, is a unix-like path, that traverses within objects, and across objects by dereferencing merkle-links. The following example demonstrates how these concepts relate to one another. The object contains information about a book, with a merkle-link that points to a different object, which contains data regarding the author.

```
{
  "title": "As We May Think",
  "author": {
    "/": "QmAAA...AAA" // links to the node above.
  }
}
```

Given the hash `QmBBB...BBB` for the object, it's possible to access the author's name via the merkle-path `QmBBB...BBB/author/name`.

3.3 Ethereum

For the purposes of this study, Ethereum smart contracts can be seen as state machines, that are deployed to the network along with an initial state, and the code necessary for future state transitions, by way of invoking public functions.

Upon deployment, they will be assigned an address, which can hence be used to interact with them. This interaction takes place, by crafting a transaction containing the target address, the sender, value of ether to be transferred, and if target is a contract, the input data

passed to the contract.

Transactions are broadcast to the network, and so-called miners propose blocks which contain a list of the previously broadcast transactions. Every other node, upon receiving a block, runs all transactions inside, and validates the computed state, against the state put forth by the miner.

Miners receive a reward in ether, the native currency of the network, for helping secure the network, and to protect against Sybil attacks[56], miners compete for proposing blocks by solving a Proof of Work[57].

As mentioned, every node in the network verifies every block, which imposes a limit on the size and frequency of blocks, which results in a limited number of slots for transactions. Users, compete for the limited slots, by sending gas (in ether) along with their transactions, which the miner earns for including the transaction in a block. Gas also acts as a deterrent for spamming the network. Miners, often employ the simple strategy of including transactions which have the most payoff.

4

Decentralized Linked Data Infrastructure

The first half of this work involves designing and implementing a decentralized linked data platform (Open Knowledge, or OK). This chapter will cover the aforementioned design and how it was implemented.

4.1 Overview

Open Knowledge relies on two open technologies. First, IPFS for the actual storage and retrieval of raw data, and Ethereum, for tracking ownership, versioning and other metadata belonging to the knowledge graph, and later on, as will be discussed, for decentralized curation of datasets.

Permissioned, centralized triplestores often store all inserted triples in a single index. However, this is not desirable in a permissionless setting where any entity has write access to the same store, meaning, entities can even publish triples that are in conflict with others already in the triplestore. Hence, in OK, knowledge graphs are not only conceptual, but they are actually stored in separate indices, that are managed by their corresponding publishing entity. The knowledge graphs are still connected by the URI scheme, and it is possible to do federated queries across multiple knowledge graphs. One can imagine knowledge graphs to be the counterpart of servers which contain a single dataset (as opposed to multi-dataset repositories).

The access control mechanism is controlled on the blockchain level, and the P2P storage layer is agnostic to access. Due to the immutable nature of IPFS, this introduces no conflict, as each modification to an existing object results in a totally new object (with a new address), regardless of who has published the modified dataset.

4.2 Storage

Each knowledge graph is indexed as a Hexastore [28] on IPFS. However, the Hexastore is not stored as a single data object, but is rather broken into smaller data nodes, which are

connected via links, forming an IPLD merkle-dag.

4.2.1 Index structure

To expand on that, each knowledge graph has a root object, with 6 keys, namely `spo, sop, pso, pos, osp, ops` where the value of each key is a link to another object containing the triples for that subindex.

Each subindex is itself a merkle-dag of depth 3. It contains the first part of the triple, with links to objects containing the second part, which in turn have links to objects containing the third part. The leaves are a simple boolean, indicating that the triple with part 1, 2 and 3 exists in the index.

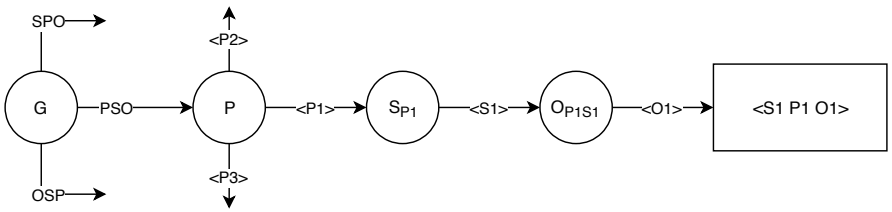

Figure 4.1: Example Hexastore Merkle-DAG for graph G.

As an example, consider graph G, as shown in figure 4.1. The figure only displays the merkle-path to the triple `<S1 P1 O1>`, via the subindex `pso`. P is the set of all predicates in G. If `<P1>` is one of those predicates, by traversing the link for `<P1>`, we arrive at the object S_{P1}, which is the set of all subjects in G for which at least one triple exists with predicate `<P1>`. In a similar manner, if `<S1>` is a subject in S_{P1}, by traversing the link, we arrive at the object O_{P1S1}, which is the set of all objects in G for which triples exist with the triple pattern `<S1 P1 ?O>`. Traversing the link for `<O1>` we arrive at the leaf object `{ "exists": true }`.

In figure 4.1, only the path for subindex `pso` is shown. However, the same triple is indexed under the other subindices. Therefore, if G has the root hash `QmAA...AA`, the merkle-paths `QmAA...AA/pso/P1/S1/O1`, `QmAA...AA/sop/S1/O1/P1`, etc. would all be `true`.

Please note that, one of the requirements of IPLD is that the keys of an object exclude a pre-defined set of special characters, including `/`. Due to the fact that URIs often contain such characters, a sanitization step must be performed to escape such characters before storing them. In Open Knowledge, all URIs are encoded using percent-encoding*, as specified in RFC3986.

4.2.2 Publishing a graph

An initial nested object O is created in memory, which will be filled with the given triples from the set T. The initial O is as follows:

Listing 4.1: Initial diff object created for publishing a graph.

```
1 { spo: {}, sop: {}, pso: {}, pos: {}, osp: {}, ops: {} }
```

Each subindex is populated with triples from T in a nested manner. For example, the populated O given $T = \{< abc >, < mbn >\}$ will be as follows:

The next step is to store the populated O on IPFS, according to the index structure. Because the data objects closer to the root of the index have links pointing to the objects closer to the leaves, storing must be done bottom-up.

First, the static object `{ "exists": true }` (henceforth L, and we denote its hash by L_h) is put on IPFS. This object only needs to be stored once and can be used for every graph. Then, all objects with height 3 are stored on IPFS, with links pointing to L. Following the

*https://tools.ietf.org/html/rfc3986#section-2.1

Listing 4.2: Populated diff object created for publishing a graph.

```
{
  spo: {
    a: { b: { c: true } },
    m: { b: { n: true } }
  },
  pso: {
    b: { a: { c: true }, m: { n: true } }
  },
  ...
}
```

example given in listing 4.2, the objects at height 3 would be `{ c: ` L_b `}` and `{ n: ` L_b `}`. Each of these objects will produce a hash, which will be used for linking in the higher levels. The same process happen for objects at every height, until we compute the hash for the root of the graph object, which itself contains links to each of the subindices (`spo, pso, ...`).

4.2.3 Updating a graph

To update an existing graph, a similar procedure takes place. The difference is that, instead of adding all the triples to a nested object O, and putting them on IPFS, it will be calculated which parts of the existing index have been modified, and O will only contain the difference between the existing graph and the new one. Then, those objects that have been modified are updated, again from bottom to top. However, when a data object down the index is updated, the links pointing to it will become obsolete. Therefore, the parents of that object in the tree also need to be updated, and change is propagated to the root. Therefore, adding even a single triple, requires 4 updates, and it will definitely result in a new root hash, thereby granting immutability to the graphs.

4.3 Retrieval

As seen in the previous section, triples of a knowledge graph are stored in the form of a merkle-dag on IPFS. Merkle-paths allow querying triple patterns, but not other features of an advanced query language such as SPARQL. It is however possible to perform a subset of all SPARQL constructs, by combining the results of several triple pattern searches. First, we will demonstrate how a simple triple pattern search could be performed, and then discuss how full SPARQL queries, either on single graphs or a federation thereof, could be executed by using triple patterns as building blocks.

4.3.1 Triple Patterns

A triple pattern, is a triple where any of the parts can be a variable instead of a concrete value. In the simplest case, it'd be possible to query the existence of a triple, that has no variable, in the knowledge graph. In this case, the merkle-path for the triple `<a b c>` would look like `QmAA...AA/spo/a/b/c` which returns `true` if the triple exists, and throws an error otherwise.

Given a graph G which has root hash G_h and a triple pattern T, the algorithm for constructing the corresponding merkle-path P and retrieving the values at this path is given below:

1. Initialize P to G_h
2. Parse T to get list of fixed and variable parts
3. Compute best subindex by bringing fixed parts first, and appending variable parts
4. Add subindex to P

5. Append values for fixed parts to merkle-path, separated by `/`

6. Fetch result (R) of P from IPFS

7. If result is nonempty, construct triples by adding the values for fixed parts to the results which were returned for the variable parts, and return them

As an example, running the algorithm over a knowledge graph which contains `[<a b c>, <f b c>]`, with `T = <?s b c>`, will result in `P = QmAA...AA/pos/b/c` and `R = [a, f]`, and the algorithm will return `[<a b c>, <f b c>]`.

4.3.2 SPARQL Queries

Although querying triple patterns and compositions thereof would suffice for some applications, it falls short for others. In order to allow SPARQL queries, Open Knowledge builds on the Linked Data Fragments framework[14] by implementing the Triple Patterns Fragments interface. By doing so, the TPF client could decompose a SPARQL query into triple patterns, retrieve the corresponding responses, and compute the final SPARQL response therefrom.

In implementing the TPF interface, a few points must be taken into account.

- TPF has been design with REST APIs in mind. The client sends HTTP requests to a centralized server, and receives HTTP responses. In Open Knowledge, the client and the implementation of the TPF interface reside on the same node, and they communicate via intra-process means.

- The given TPF interface implementation runs on a server and fetches data from a local database, whereas in Open Knowledge, triples are stored across peers, and in case the

required triples are not replicated locally, each triple pattern query is translated into requests to fetch triples from other peers.

- Pagination and control functions such as `nextPage`, are a requirement of the TPF interface. Currently pagination is done after fetching all of the triples matching a pattern. This can however be further optimized by storing size metadata in the indices.

Federated SPARQL queries are performed in a similar manner, by utilizing the TPF interface. However, TPF requests the result of triple patterns from servers via HTTP, whereas in Open Knowledge, all triple pattern queries are done simply over different graphs which exist on the same P2P filesystem.

4.4 Smart Contracts

So far we've seen the structure of indices, how knowledge graphs are published and updated, and how queries are performed over graphs. Each graph G is identified by the multihash of the root of its index, and updating the graph results in a completely new and unpredictable root hash. As a result, data consumers need a means for tracking the history of changes to G and consequently its root hashes, in order to be able to perform queries.

In this section, two smart contracts, `Graph` and `SimpleRegistry` will be introduced, which facilitate tracking the history of graphs and their metadata, and improving findability by naming them.

4.4.1 Graph

The `Graph` smart contract is meant to represent a single dataset, maintained by a single entity. It tracks the history of the graph, stores relevant information such as version, and points to additional metadata that the author wishes to attach to their dataset.

When creating a new knowledge graph, the publisher must publish the RDF triples on IPFS, as outlined in the previous section, and deploy an instance of the `Graph` contract, providing the root hash of the index as input. The deployed instance has a permanent address, which they can distribute to data consumers. Consumers can then query the Ethereum blockchain to fetch the state of the aforementioned contract instance, find the current root hash which they can use to perform queries. To update the knowledge graph, they update the index on IPFS, and make a transaction to the contract, providing the new hash as input. Consumers who are subscribing to events emitted by Ethereum, will be informed of the new root hash.

The smart contract holds the following state fields:

Listing 4.3: State of the Graph contract in Solidity. For brevity, the rest of the contract has been omitted.

```
contract Graph {
    address public owner;
    bytes32 public id;
    bytes32 public root;
    bytes32 public metadata;
    bytes32 public license;
    uint public version;
}
```

Ownership

In listing 4.3, `owner` refers to the Ethereum account who deployed the smart contract. From then on, only `owner` is able to modify the state of *c*, but ownership can easily be transferred to other accounts by submitting a transaction and invoking the method `setOwner`.

Please note there's no inherent difference between the address of an external account and that of a contract. Therefore, the `owner` could be the address of a multisignature contract, enabling complex signature schemes, such as `(t, n)-threshold`, for updating the state.

History

The field `root` is an IPFS hash which points to the root of the knowledge graph's hexastore index. When *G* is updated, `owner` sends a transaction to *c*, updating `root`. This removes the need for a side-channel to announce new versions of *g*, and the need for maintaining a list of previous `roots`, as Ethereum full-archive nodes by default store all of the previous states.

Furthermore, versions of *G* are automatically tagged by an auto-increment `version` field, which can be used to query specific versions of *G* without referring to the full IPFS hash in SPARQL, as will be discussed in the next section.

Metadata & Attribution

The smart contract also keeps an optional field `metadata`, which in a similar manner to `root`, is an IPFS hash. The IPFS object identified by `metadata` should contain additional information about the knowledge graph, ideally in an IPLD-compatible format. The structure and semantics of `metadata` has been left to individual graphs, but they could potentially include information about the authors, citations to other graphs and a website link for further

information.

4.4.2 Simple Registry

The `SimpleRegistry` contract (R) acts as a knowledge graph name registry, and a list for data consumers to find knowledge graphs. Without it, data consumers would have to know the address for every knowledge graph, and would have to specify that address in their queries. R allows registering graphs under a unique name (first-come first-served), and later on request the contract address for a certain graph with its name.

It's important to note that, this contract is also openly available, and an instance of it can be deployed by any party. Data producers can decide which registry they want to be a part of.

`SimpleRegistry` also has a convenience method `newGraph(bytes32 _name)` for deploying an empty `Graph` contract and registering a name for it in one transaction.

4.4.3 Upgradability

The code for a smart contract is immutable, and cannot be modified after deployment. There are, however, circumstances which necessitate modifications to an existing contract, such as when a bug has been discovered, or when upgrading can improve interoperability by adopting a standard that came into being after the contract was deployed. With those cases in mind, the `SimpleRegistry` employs a work-around that allows upgrading the contract. This has to be done, however, with care, and only with the agreement of all parties involved.

The mechanism makes use of a proxy contract, in addition to contract which contains the logic. The proxy contract acts as a wrapper to the logic contract, and is the interface with which the users interact. It stores the state, and a pointer to the current logic contract, and

it delegates every incoming transaction, by means of the `DELEGATECALL` EVM opcode to the logic contract. During an upgrade, a new instance of the logic contract is deployed, and the proxy is invoked to update its pointer to the new contract.

4.5 SDK

The Javascript SDK brings the aforementioned parts together and provides a uniform and simple interface to developers who want to include Open Knowledge in their applications, effectively hiding much of the complexity. It has been designed in a way that would allow swapping the underlying technologies. It implements the procedures that were detailed in the previous sections for creating, updating and querying knowledge graphs.

The implementation details of the SDK are out of the scope of this work, however to showcase how a developer would use it, a few examples are given below:

Listing 4.4: SDK example for creating a new graph, and adding triples

```
1 let manager = await ok.newGraphManager('myGraph')
2 let tx = await ok.addTriples([['subject', 'property', 'object']], 'myGraph')
```

Listing 4.5: SDK example for querying simple triple patterns

```
1 let res = await ok.getTriples(null, 'http://dbpedia.org/ontology/influenced', null, '
      dbpedia')
```

To give an overview of how the different parts come together, we'll investigate listing 4.6 briefly. The SDK validates the SPARQL query, and extracts the knowledge graph on which the query should be executed. Please note the `FROM <openknowledge:dbpedia>` on line 4. This is a valid SPARQL IRI, which specifies the scheme as `openknowledge` and the name of

Listing 4.6: SDK example for SPARQL Query

```
let res = await ok.execute(‘
  PREFIX dbr: <http://dbpedia.org/resource/>
  PREFIX dbo: <http://dbpedia.org/ontology/>
  FROM <openknowledge:dbpedia>
  SELECT *
  {
    dbr:Lucky_Starr_and_the_Big_Sun_of_Mercury dbo:author ?o.
    ?s dbo:influenced ?o
  } LIMIT 15
‘)
```

the graph.

If the graph wasn't cached locally, the SDK queries the `SimpleRegistry` contract to find the address of `dbpedia`'s respective `Graph` contract. It then consults the `Graph` contract to fetch the root of the hexastore index on IPFS.

At this point, the SDK can construct the Triple Pattern Fragments client, and ask IPLD for the decomposed triple patterns, combine the results and return them to the caller in pages, according to the algorithm specified in sections 4.3.1 and 4.3.2.

Previous versions

By default, when performing queries over a named knowledge graph as in listing 4.6, the SDK uses the latest version of the graph. This could fail in some cases:

- If a the knowledge graph has been updated just shortly before, it might take a while until there are enough replicates in the network to respond with the requested triples.
- Or, due to a mistake, or on purpose, the root that's submitted to the Graph contract points to an invalid data structure.

If the query fails over the most recent version, the SDK falls back to previous versions.

In addition to the fallback behaviour, the SDK supports running queries over specific versions of a graph, by adding the version to the `FROM` part of the query, as in the example `FROM <openknowledge:dbpedia:2>`, where `2` is the version.

5

Scalable Curation

Low barrier to entry within a decentralized infrastructure, could potentially increase the already high growth rate of the number of datasets being published in the web of data. This poses a findability issue for data consumers. With a growing number of datasets, it becomes increasingly costlier for consumers to find knowledge graphs suitable for their purposes, and

upon finding graphs, for them to gauge their quality. This highlights the need for scalable curation mechanisms, which we will try to address in the following chapter.

5.1 Overview

Free participation and censorship-resistance in Open Knowledge (refer to chapter 6 for discussion) has two sides. On the one hand, these characteristics ease the publication of useful and high-quality data for everyone. On the other hand, they make the infrastructure prone to being flooded with low-quality and less relevant data.

Any entity can easily create as many knowledge graphs as desired. The gas costs act as a deterrent for spamming the network. Even so, the number of legitimate graphs could potentially increase to be high enough, as to make the cost of finding suitable graphs among them non-trivial, assuming there exists a channel from which consumers can find the address of all graphs.

However, because `Graph` contracts have a unique and persistent identifier, namely the address of the contract, it's possible to create public lists (or collections) of graphs that are relevant for a given purpose or satisfy certain quality requirements, which consumers can refer to. Upon finding a graph in such a list, consumers can provide the graph contract address as input to the SDK (section 4.5), and execute queries against it.

Relevance is a context-dependant quality, and each context might call for a different set of trade-offs. Therefore, there likely won't be a single most-relevant list, but rather a plethora of lists, maintained through various mechanisms. Each knowledge graph could be listed in multiple of collections, and a global quality score of such a graph could be measured by taking the average score of the graph across collections.

The goal is therefore to consider curation mechanisms, the output of which is a list of valid and relevant knowledge graphs for data consumers. In the following sections, mechanisms with different trade-offs, ranging from a curation by a committee of experts, to a crowd-sourced mechanism which is distributed and incentivized will be discussed.

5.2 Single-entity curation

5.2.1 Manual

An entity E (e.g. expert committee) assesses knowledge graphs manually and maintains a collection of the relevant/high-quality ones. E defines what is considered relevant or high-quality, and the rules of the selection process. Often, data publishers would have to submit their datasets to be assessed by E. Data consumers subscribe to the list based on prior trust of E. In large scale, this approach could become prohibitively costly.

5.2.2 Algorithmic

An entity E selects a relevance function r, a set of rules R, and designs a program which automatically selects the graphs in the curated list according to their relevance score ($r(g)$) and R. It remains to be seen if such algorithmic assessments can determine quality or relevance. As has been shown, they can however be used for pre-filtering by detecting "bad smells" [55] with a low cost.

The `SimpleRegistry` contract as specified in section 4.4.2, has no relevance function, and only one rule, namely that no other knowledge graph exists with a given name.

Any entity e can add knowledge graphs to the list, but no entity can remove graphs from the list. Effectively, `SimpleRegistry` is an append-only log, which due to low costs of including a graph, provides no filtering of graphs by itself. But already there's a shift from the two previous mechanism, and the shift comes down to the fact that no single entity has control of the items in the list.

`SimpleRegistry` can further be extented, in a way that every entity i has a reputation w_i, and they can vote for knowledge graphs in the list with weight w_i. Therefore, this mechanism has one rule for including graphs, which is that no other graph with the given name exists in the list, and the relevance function $r(g) = \sum_{i=0}^{N} w_i$ (or a normalized variant thereof), where $w_i = 0$ if i has not cast a vote for g. Simple registry can be seen as a special case of reputation-based registry, where $\forall w_i : w_i = 0$.

In this theoretical method, any entity can add a graph to the registry, and the graph would be assigned a relevance score, based on the votes of independent entities who have a good sense of what is relevant in the given context (due to their reputation weight). The caveat however is that assigning reputation to entities in a public blockchain is an open problem*. A trusted entity needs to be introduced, which attests to the reputation of other individuals.

*https://github.com/ethereum/wiki/wiki/Problems

As seen in chapter 4, Open Knowledge only stores the root hash of the index stored on IPFS, and not the index itself in the Ethereum smart contract. The main reason behind that, is transaction costs that are incurred due to storage.

The smart contract has no way of verifying whether the hash h actually points to a valid knowledge graph stored on IPFS. However, using Merkle proofs, or a zk-SNARK[58] proof, it'd be possible to prove to the smart contract, that a valid graph index would result in h as root of the index. Although verifying this proof is much cheaper than sending or storing the whole graph index, the transaction cost is still high enough to make it infeasible to do for every graph update.

However, if we have the expensive method $\mathcal{M}$ for verifying the validity of a graph on-chain (e.g. a zk-SNARK verifier), by utilizing prediction markets, it'd be possible to check a larger number of graphs for validity, with only a smaller subset of them needing to revert to $\mathcal{M}$ for verification [23].

Any entity e_1 could claim that a given graph g is invalid by creating a bet of size x in the prediction market. If e_2 doesn't agree with g being invalid, they'd put a bet of size y on the opposite side.

If, after a pre-specified period has passed, no other entity has challenged the bet, g would hence be considered as invalid. Otherwise, verifying via $\mathcal{M}$ the winning side is determined. Each entity in the winning side is rewarded proportional to their bet, a part of the bets of the losing side.

The process can be further optimized to deter incorrect betting and volume manipula-

tion, by having the amount won to be only 75% of the amount lost. The other 25%, could for example be distributed to producers of valid graphs.

The rationale here is that, verifying the validity of a graph is much cheaper done off-chain, than on-chain. Therefore, users would be incentivized to "fish" invalid graphs. They are disincentivized to bet against a valid graph, because others can challenge the bet in the market, and an on-chain verification would result in the loss of their bet.

This mechanism, could also be used as a pre-filter for other mechanisms.

5.5 Token-Curated Registry

Token-curated registry (hereafter TCR) [24] is a mechanism, in which rational actors are incentivized to maintain a decentrally-curated list. As the name suggests, TCRs rely on a native token, which has a value relative to another base currency (fiat currencies, such as EUR). Apart from consumers of the curated list, which desire a high-quality list of knowledge graphs, other actors require tokens to interact with the TCR.

5.5.1 Actors

Actors of a TCR include candidates, voters and challengers. Please note that, these characterizations are not mutually exclusive. A candidate, is an actor, who wishes to add a graph to the list, and stakes N tokens along with the application. Challenger, is an actor, who believes the item that a candidate proposed, does not belong in the list, and is willing to stake N tokens to challenge the application. When a challenge occurs, a voting period starts, during which, token holders can cast a vote, either for or against the item in question. Votes are weighted proportional to the number of tokens the token holder specifies. The tokens would not be

spent during a vote. After the voting period comes to an end, the side with most token-weighted votes wins, and depending on the outcome, either the candidate or the challenger loses a portion of their stake, and this portion is split among the winners, in proportion to the number of tokens they participated with.

5.5.2 Rewarding honest behaviour

The rationale behind TCRs is that, rational voters who seek to increase their long-term profit, would only vote to accept items that have a relative higher quality, which increases usefulness among consumers, resulting in more demand among candidates to be on the list, that in turn increasing the value of the native token with respect to the base currency. This is in addition to their short-term benefit of being rewarded with more tokens, if they vote for high-quality items.

5.5.3 Disincentives

The risk associated with losing the stake, disincentivizes candidates to apply for a graph, they're aware has low quality or is invalid. At the same time, challenging a high quality graph also comes with the risk of losing a portion of challenger's stake. If this was not the case, participants would have been incentivized to challenge every application, effectively requiring a vote on every application, and thereby reducing the efficiency of the mechanism.

Vote-splitting

The aforementioned spec failed to address the "nothing at-stake" problem for voters, or in particular, the "vote-splitting" issue, in which, a rational strategy for voters could be to split

their tokens in half, and vote for both side, thereby earning revenue regardless of the outcome of the vote, and without putting in any effort. TCR 1.1 [†] addresses this issue, by slashing a portion of the minority bloc's tokens, and adding it to the rewards of the majority's bloc.

5.5.4 Commit-reveal voting

Due to Ethereum transactions being public, during a voting period, voters can see the current tally, and vote with the majority, without inspecting the item in question. This can be prevented, by splitting the voting period into two phases: first, all voters make a cryptographic commitment to a vote, after the commit period has come to an end, everyone must reveal their vote, by submitting the secret used to make the commitment. Consequentially, the tally of the votes is unknown by everyone other than the voter until the end of the commit period. This effectively prevents voters from basing their decisions on how others are voting.

5.5.5 Listing item status

Graphs are added to the list, either if they face no challenge after application, or if they are challenged, and voters vote for inclusion of the graph. The stake, which is a requirement of applying to the TCR, remains locked while the item is in the list. The candidate, can, at any moment withdraw the stake, thereby removing the item from the list.

Furthermore, even after a graph has been listed, it can be challenged and therefore removed from the list. This is inevitable, because an append-only list, could grow large enough to lose its usefulness, and as such, when higher quality graphs are added to the list, lower quality graphs can be challenged and removed, in order to maintain a limited number of slots in

[†]https://medium.com/@ilovebagels/token-curated-registries-1-1-2-0-tcrs-new-theory-and-dev-updates-34c9f079f33d

the list.

Challenging a graph after it has been listed could furthermore be necessary in cases where, after an update the new updated graph no longer adheres to the standards of the list.

5.5.6 Variants

The mechanism discussed until now, would result in an unordered list of graphs, potentially with a finite number of graphs. Various other mechanisms have been proposed, that build on the basic idea of a TCR to make it suitable for different applications.

Layered TCR

In a layered TCR [‡], items first apply for admission into the L_0 TCR, which has a lower quality bar. At any point, an item in level L_i, can apply for admission into L_{i+1}, and can be challenged to drop to L_{i-1}.

Nested TCR

In a nested TCR [§], a tree of TCRs is formed, in which TCRs in the leaf nodes point to graphs, but other nodes in the tree have other TCRs as items of their list. In the case of a linked data platform, the hierarchy, could represent categories of data, with TCRs further down the tree having more specialized categories.

[‡] https://blog.oceanprotocol.com/the-layered-tcr-56cc5b4cdc45
[§] https://medium.com/@DimitriDeJonghe/curated-governance-with-stake-machines-8ae290a709b4

5.6 Data Quality

Missing from the aforementioned curation mechanisms is a definition of data quality, or fitness for use. Data quality assessment processes would differ among mechanisms, depending on their requirements and purposes. When quality is to be determined by a select committee of experts, systematic assessments [16] could be employed, which is subject to prohibitive costs in larger scales.

Whereas in the more decentralized mechanisms, such systematic assessment by all participants is unfeasible, specially that intuitively the probability of being an expert decreases as participation becomes more decentralized. On the other hand, such methods are suitable for large scale, and they incur lower costs. To further improve on these methods, a promising direction is the use of automatic syntactic validation and detection of "bad smells" [55] to help participants in assessing quality.

Furthermore, systematic assessment by experts, and decentralized usefulness assessment can be combined, as they've been shown [17] to be complementary.

5.6.1 Implementation

An open source implementation of the smart contracts necessary for a generic TCR has been made available by the original proposer of TCRs [¶]. In order to integrate TCRs with Open Knowledge, a Javascript library [‖] has been implemented, which handles most of details necessary in using the TCR smart contracts, such as state management. The `tcr.js` library, has been integrated into the `open-knowledge` Javascript SDK, to provide a simple interface to

[¶] https://github.com/skmgoldin/tcr
[‖] https://github.com/planet-ethereum/tcr.js

users of Open Knowledge to interact with a TCR, by creating new graphs, applying for them to be listed, challenging graphs, etc. The items in the TCR, are addresses of `Graph` (section 4.4.1) contracts.

6

Results & Discussion

In this chapter, the results of a benchmark performed on the prototype implementation will be presented. Next, qualitative characteristics of the design, including the FAIR principles, will be discussed. In the end, a potential attack vector on the TCR curation mechanism will be outlined.

6.1 Results

To verify the correct functionality of the spec, the source code (https://github.com/sina/open-knowledge) includes a test suite, which comprises of three categories of tests. First, each Solidity smart contract has an accompanying test, which checks the initial state of the contract, as well as correct state transitions upon invocation of the contract methods. Second, modules of the SDK have accompanying unit tests, which aim to verify the functioning of that module in isolation. For this purpose, two key components with which many of the modules interact, namely IPFS and Ethereum, have been mocked with an in-memory object that adheres to the IPFS interface and an in-memory deployment of the smart contracts. Finally, the integration test cases test the system as a whole by storing triples and querying the stored triples.

To gain further insight into the consequences of the design, specifically the indexing design, and to highlight potential bottlenecks for future work, we generated a WatDiv [59] dataset with scale 1 which contains 107665 triples, stored the dataset on Open Knowledge and performed the 20 "basic testing" queries provided in the WatDiv (v0.6) packaging, comprising of linear queries (L), star queries (S), snowflake-shaped queries (F) and complex queries (C). The benchmark was executed on a personal laptop with Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz, SSDSA2BW16 disk and 8 GB of memory, running a Linux kernel (v4.19.1). By default the SDK maintains a cache of results fetched from IPFS. To measure query execution times, each query has been executed 5 times with a warm cache, and 5 times with a cold cache. Results of the queries have been compared for correctness against the ARQ engine *.

Storing the dataset comprises of two main phases, constructing the index tree locally and

*https://jena.apache.org/documentation/query/index.html . Accessed: November 2018

storing the tree on IPFS from leaves to the root. Constructing the tree locally took 1503 ms, and storing it on IPFS took 18.52 minutes and translated into 318972 IPFS `PUT` requests.

Figure 6.1 displays execution times measured for the aforementioned queries in a logarithmic scale. The difference between *default* and *w/o cache* traces is in caching the results of IPFS `get` requests in the SDK. Table 6.1 outlines measurements of metrics during each query, providing additional insight into factors potentially influencing the execution times. *Triple patterns* denotes the number of triple patterns each SPARQL query is decomposed into by Triple Pattern Fragments client, *IPFS gets* is the number of `GET` requests to IPFS, *Repeated paths* is the number of paths that had been requested from IPFS during the same query and *returned triples* denotes the total number of triples that have been returned from IPFS to construct the final SPARQL result.

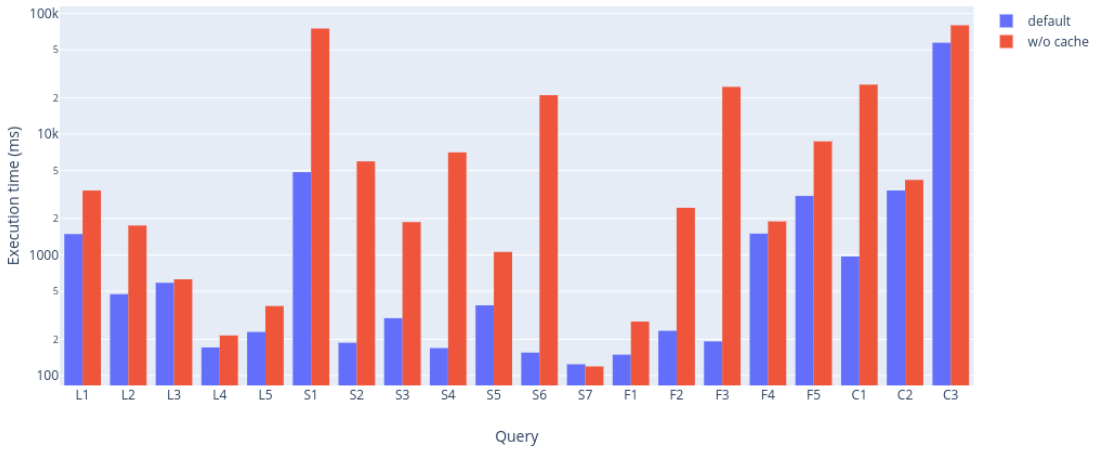

Figure 6.1: Query execution times.

Query	Triple patterns	IPFS gets	Repeated paths	Returned triples
L1	131	403	79	1897
L2	26	225	3	387
L3	27	253	1	1107
L4	11	34	1	39
L5	13	296	45	297
S1	375	6849	152	8357
S2	13	990	1	1205
S3	14	305	10	664
S4	9	725	1	752
S5	16	220	1	242
S6	6	1405	3	1510
S7	3	1424	13	1499
F1	11	1743	10	2201
F2	27	2135	104	2235
F3	9	2212	1	3552
F4	271	5269	3748	6565
F5	363	68196	62629	72297
C1	51	8343	6057	10120
C2	183	17895	14906	36216
C3	3672	6851	3148	53821

Table 6.1: Performance metrics for WatDiv basic testing queries.

6.2 Discussion

Open Knowledge (OK) can be seen as a global linked open data repository which facilitates storing knowledge graphs and retrieving triples from either single graphs or a multitude of them. To validate its functionality with larger datasets and to highlight potential emerging bottlenecks in the design for future work, a benchmark was performed as described in the previous section. In particular, the metrics shown in table 6.1 point to the number of decomposed triple patterns and IPFS requests as a potential factor that correlates with execution time. The *repeated paths* metric further reemphasizes the benefits of a cache for intermedi-

ate results retrieved from IPFS. In the rest of this section, the characteristics of OK will be compared with centralized infrastructures qualitatively.

6.2.1 Characteristics

Free participation Any entity is able to query knowledge graphs. Likewise, any entity is able to store knowledge graphs. The transaction fee for deploying a `Graph` contract can be seen as the only barrier to storing graphs, which can be avoided by only publishing the graph on IPFS and not deploying a contract. Consumers would still be able to use the graph, if they have access to the root hash.

Censorship-resistance No entity can prevent others from storing or querying knowledge graphs.

Fault tolerance Open Knowledge inherits the fault tolerance properties of its underlying technologies, i.e. IPFS and Ethereum.

Availability In a centralized storage setting, data publishers run a server, the availability of which equals the availability of the dataset. In a P2P storage setting, availability of a dataset depends on the availability of all the peers which store a replica of the dataset. Availability is typically measured by the percentage of uptime of a given resource during a period. If the probability of one of these peers being available at moment t during a given period (normalized uptime) is $P_i(t)$, the availability of the dataset can be calculated to be $P(t) = 1 - \prod_{i=0}^{n}(1 - P_i)$. Therefore, if the data publisher runs an IPFS node on the same server and pins the dataset, the availability of the dataset published on Open Knowledge would be equal to that of a centralized storage scheme, if no other

replica exists, and strictly greater, if at least one other replica with $P_i(t) > 0$ exists.

Secure Versioned URIs The infrastructure allows for various URI schemes with different trade-offs. Using root hashes to refer to datasets ensures data authenticity for retrieval. Furthermore the underlying data behind a root hash never changes, which is a key feature in Cool URIs [†]. In contrast, changes in the data behind a HTTP-based URI, don't get reflected in the URI by default. Furthermore DNS domains can expire, and be transferred to different owners, or face other attack vectors such as DNS spoofing. On the other hand, root hashes don't have the simplicity feature of Cool URIs. If that is desired, data publishers can assign ENS [60] domains to their datasets which still ascertains data authenticity (as they point to root hashes themselves which will ultimately be used to retrieve data) and they keep a history of the content they refer to, and are DoS-resistant.

Access Control A `Graph` contract can only be updated by the respective `owner`. Correctness of this behaviour is verified using unit tests, which can be found in the `test/` directory of the repository.

6.2.2 FAIR principles

The widely accepted FAIR guiding principles [10] put forth a set of metrics that aim to measure data management strategies. A FAIR data repository leads to better knowledge discovery, evaluation and reuse.

Publishing data on Open Knowledge instead of a server improves FAIRness on several metrics, particularly with regards to Findability, Accessibility and Interoperability. Data

[†]https://www.w3.org/TR/cooluris/

publishers can further improve FAIRness by adopting community endorsed standards for the metadata of the graph. Here, we discuss in detail how publishing on Open Knowledge affects each of the metrics. For each of the core principles, the specific metrics are listed, along with a discussion of how Open Knowledge fares for that metric.

Findable

1. *(meta)data are assigned a globally unique and eternally persistent identifier.*

   The root hash of the index of a specific version of a document stored on IPFS, acts as a globally unique and eternally persistent identifier. Furthermore, the `Graph` contract address, acts as a globallly unique and eternally persistent identifier for the metadata of a knowledge graph.

2. *data are described with rich metadata.*

   A `Graph` contract contains itself the most necessary metadata regarding a knowledge graph, and it furthermore stores an IPFS multihash which points to a document with all the complementary metadata of that knowledge graph, such as provenance and licensing information. The exact format and attributes of such document is left open for publishers, as long as the format adheres to IPLD.

3. *(meta)data are registered or indexed in a searchable resource.*

   `Graph` contracts are indexed and searchable on a public distributed ledger technology. The complementary metadata is also stored on IPFS, it is indexable by its hash, and searchable by IPLD.

4. *metadata specify the data identifier.*

The `Graph` contract stores the multihash pointing to the most recent version of the knowledge graph. Furthermore, identifiers for previous versions of the knowledge graph are also publicly available, as Ethereum has a public state history.

Accessible

1. *(meta)data are retrievable by their identifier using a standardized communications protocol.*

   IPFS and IPLD which are used for storing and retrieving data are standardized, but the specs are not yet fully finalized. The specs, and their stability, can be tracked at `https://github.com/ipfs/specs` and `https://github.com/ipld/specs`.

   There are various ways of retrieving the metadata stored on the `Graph` contract on Ethereum. The standardized JSON-RPC API of an Ethereum client can be found at `https://github.com/ethereum/wiki/wiki/JSON-RPC`.

2. *the protocol is open, free, and universally implementable.*

   Both protocols mentioned above are open, free and universally implementable, and there are already multiple implementations of each.

3. *the protocol allows for an authentication and authorization procedure, where necessary.*

   Modifying an existing `Graph` contract is protected, and access is granted only to the entity controlling the private key (in case of an external account, otherwise the smart contract that owns the graph) of the Ethereum account which is the current `owner` of the contract. Otherwise, publishing new data, or querying existing data is purposefully left unauthenticated and open for free participation. If needed however, the data on IPFS can be encrypted, and a list of public keys that are allowed access, added to the

metadata of a knowledge graph, to limit read access to those who control the private keys for the aforementioned whitelisted public keys.

4. *metadata are accessible, even when the data are no longer available.*

   Ethereum smart contracts will exist as long as there exists at least one full node, unless they call the `selfdestruct` opcode. Since the `Graph` contract doesn't call this opcode, their lifetime is that of the Ethereum network.

Interoperable

1. *(meta)data use a formal, accessible, shared, and broadly applicable language for knowledge representation.*

   The Hexastore model for the knowledge graph stored on IPFS, assumes a triple-based data structure, and the JS library includes a parser for various RDF representations such as N3, turtle, etc. The exact format of the metadata document is left to data publishers, but due to the requirement of it being an IPLD-supported format, it can be selected from a variety of widely-used formats such as JSON, YAML, Protobuf, XML and RDF.

2. *(meta)data use vocabularies that follow FAIR principles.*

   The platform is agnostic to the vocabularies used in knowledge graphs. It's recommended however that data publishers use standards that are widely accepted among the scientific community, such as the ones outlined by the FORCE11 Data Citation Implementation Working Group for data citation [61] in the metadata document of a knowledge graph.

3. *(meta)data include qualified references to other (meta)data.*

   Similar to other knowledge graphs, documents stored on Open Knowledge can use IRIs to refer to other documents. Data publishers can furthermore add references to other knowledge graphs in the metadata document, e.g. by referring to their `Graph` contract address or their IRI.

Reusable

1. *meta(data) have a plurality of accurate and relevant attributes.*

   The exact attributes included in metadata is left to data publishers. As mentioned before, it is recommended to use standards that are widely accepted among the community.

2. *(meta)data are released with a clear and accessible data usage license.*

   The `Graph` contract stores the license for the knowledge graph, as specified by the data publisher.

3. *(meta)data are associated with their provenance.*

   Data publishers are expected to include provenance information in the metadata document of their knowledge graph.

4. *(meta)data meet domain-relevant community standards.*

   The platform itself is agnostic to metadata and data standards.

6.3 Analysis of TCRs

When evaluating the security properties of TCRs, we encountered a theoretical attack [‡] on SchellingCoin [§] which can be extended to TCRs. A description of this attack is outlined for TCRs in the rest of this section.

6.3.1 P+ epsilon Attack

If instead of the uncoordinated choice model, TCRs were to be analysed under the stricter bribing attacker model, a new attack vector opens up. Bribing attacker model is similar to uncoordinated choice model, in which all participants are assumed to have separate incentives and do not coordinate together, but it also assumes the possibility that there's an attacker with a budget, who is willing to pay participants if they take certain actions.

In this case, if the reward for voting honestly is P, the attacker could offer a reward of $P+\varepsilon$ for voting X, which changes the payoff function of rational agents, and incentivizes them to vote X, which has a higher payoff. If 51% of participants are rational, and vote X, they will win, and the attacker has managed to change the result of the vote, with 0 cost.

A requirement for this attack is that participants can prove they have voted as the attacker demands. This is possible, because after votes are revealed, nodes can determine how any account has voted by building a history of the EVM events the TCR contract has emitted.

Not only an attacker could observe how a participant has voted, verifying the vote of a participant and paying out the reward can be done trustlessly, as to completely eliminate the need for trust between participant and attacker, which makes the attack easier.

[‡] https://blog.ethereum.org/2015/01/28/p-epsilon-attack/ . Accessed: November 2018

[§] https://blog.ethereum.org/2014/03/28/schellingcoin-a-minimal-trust-universal-data-feed/ . Accessed: November 2018

The attacker would create a smart contract which can verify Modified Merkle-Patricia (MPT) trie proofs of the inclusion of an event log in one of the most recent 256 blocks. The limitation of 256 blocks is due to the fact that smart contracts can access the block hashes of only the last 256 blocks, given the block number. Then, the participant, after having voted X, generates the MPT proof, and submits it to the smart contract along with the block header in which it was included. The block header contains the hash of the receipts trie, which contains all of the event logs. Therefore, the smart contract can verify the proof that L is included in *ReceiptsTrie* with root hash *ReceiptHash*, and test that against the *ReceiptHash* provided in the block header. Finally it should be verified that hash of the block header sent by participant is indeed the hash of one of the last 256 blocks. After verification, the smart contract can payout the reward.

Verifying claims trustlessly incurs a transaction fee for the participant. In the case that expected reward ε is higher than estimated transaction fee, every participant is incentivized to vote X and prove that to the attacker.

7

Conclusions & Future Work

In this dissertation, we proposed a novel architecture for a fully decentralized linked data infrastructure, which has a very low barrier to entry, is censorship-resistant and benefits from fault-tolerance properties of its underlying open technologies. The availability of datasets published on Open Knowledge is dependant on their demand, but is expected to be equal or higher than a centralized storage scheme in most cases. By publishing their knowledge graphs

on this infrastructure, data producers can improve the FAIRness of their datasets. In order to ensure reusability, producers must take care however to adopt community-wide standards to describe the metadata, as the infrastructure doesn't enforce how metadata is specified.

On the other end of the spectrum, data consumers have open and free access to all datasets, they have access to the metadata of datasets and can query specific versions, and generally benefit from a FAIRer dataset. Data consumers are not limited to human agents, and the aforementioned apply likewise to machine agents. Due to immutability of each version of a dataset, consumers can cache the data objects they interact with long-term, and perform queries even while offline. By replicating these data objects, they are at the same time contributing to the availability of those datasets.

In addition to the platform, we explored two mechanisms which allow a community to come to consensus over a collection of datasets which they find relevant or high-quality in a distributed manner, by utilizing smart contracts to align the incentives of participants. As observed, due to their complexity, TCRs have a bigger attack surface, and although some of the attacks have been covered and made economically expensive by later revisions, we believe more testing and analysis are still required before it is ascertained they are not vulnerable in a production setting. Specifically, under the bribing attacker model, we looked at the P+ epsilon attack. Finding solutions for the attack are left to future work.

To better understand the trade-offs of a decentralized infrastructure, we plan to experiment with various indexing schemes, which could potentially reduce number of requests to IPFS, thereby improving query performance. Implementing the platform in a low-level, high-performance language such as Rust, which is also accessible in web applications by compiling to WASM, optimizing it, and then evaluating the retrieval performance against other

P2P and centralized RDF datastores could also paint a better picture on how a decentralized infrasturcture could compare with a centralized one.

IPFS replicates a document on every node that interacts with it. Therefore, more popular knowledge graphs are expected to be highly replicated. However, IPFS doesn't guarantee persistence. If the node that published a document goes offline, and there's no other replica, that document won't be accessible until the node comes back online. This might lower accessibility for knowledge graphs that have less demand. Future works can improve on this by incentivizing nodes to replicate pieces of data and asking them to provide *proof-of-replication* [62].

Size of knowledge graphs influence publication, retrieval and query performance, replication rates and even the efficacy of the curation mechanisms. Hence evaluating the effects of graph size can be an interesting direction for future works. We hypothesize that smaller but more interconnected graphs lead to better results for the aforementioned metrics. Smaller graphs can be maintained better, even by entities with less resources, they have lower index size which improves retrieval performance, consumers are more likely to replicate specific graphs they interact with regularly as opposed to a single large graph containing all triples. Furthermore, quality assessment of smaller graphs have lower costs which could improve TCR efficacy, as if the cost of assessment is higher than a threshold, participants might be incentivized to vote randomly, or not vote at all when a dataset is challenged.

The URI scheme used in the platform can further be extended to express not only graph names and integer versions, but also specific versions of datasets (by specifying the root hash of their index tree), and even specific triple parts. One other promising direction is using ENS, which is resistant to DoS and man-in-the-middle attacks, for naming knowledge graphs.

References

[1] W. W. W. Consortium *et al.*, "Rdf 1.1 concepts and abstract syntax," 2014.

[2] T. Heath and C. Bizer, "Linked data: Evolving the web into a global data space," *Synthesis lectures on the semantic web: theory and technology*, vol. 1, no. 1, pp. 1–136, 2011.

[3] T. Berners-Lee, "Linked data - design issues." https://www.w3.org/DesignIssues/LinkedData.html, 2006. Accessed: October 2018.

[4] S. Harris, A. Seaborne, and E. Prud'hommeaux, "Sparql 1.1 query language," *W3C recommendation*, vol. 21, no. 10, 2013.

[5] "Linkingopendata." https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData, 2010. Accessed: October 2018.

[6] S. Auer and S. Hellmann, "The web of data: Decentralized, collaborative, interlinked and interoperable," in *Proceedings of the 8th International Conference on Language Resources and Evaluation (LREC-2012)*, 2012.

[7] "The linked open data cloud." https://www.lod-cloud.net, 2010. Accessed: October 2018.

[8] R. Cyganiak, H. Stenzhorn, R. Delbru, S. Decker, and G. Tummarello, "Semantic sitemaps: Efficient and flexible access to datasets on the semantic web," in *European Semantic Web Conference*, pp. 690–704, Springer, 2008.

[9] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao, "Describing linked datasets.," in *LDOW*, 2009.

[10] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific data*, vol. 3, 2016.

[11] I. Ermilov, M. Martin, J. Lehmann, and S. Auer, "Linked open data statistics: Collection and exploitation," in *International Conference on Knowledge Engineering and the Semantic Web*, pp. 242–249, Springer, 2013.

[12] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan, "Observing linked data dynamics," in *Extended Semantic Web Conference*, pp. 213–227, Springer, 2013.

[13] E. Rajabi, S. Sanchez-Alonso, and M.-A. Sicilia, "Analyzing broken links on the web of data: An experiment with dbpedia," *Journal of the Association for Information Science and Technology*, vol. 65, no. 8, pp. 1721–1727, 2014.

[14] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert, "Triple pattern fragments: a low-cost knowledge graph interface for the web," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 37, pp. 184–206, 2016.

[15] H. Van de Sompel, R. Sanderson, M. L. Nelson, L. L. Balakireva, H. Shankar, and S. Ainsworth, "An http-based versioning mechanism for linked data," *arXiv preprint arXiv:1003.3661*, 2010.

[16] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, and S. Auer, "Quality assessment for linked data: A survey," *Semantic Web*, vol. 7, no. 1, pp. 63–93, 2016.

[17] M. Acosta, A. Zaveri, E. Simperl, D. Kontokostas, S. Auer, and J. Lehmann, "Crowdsourcing linked data quality assessment," in *International Semantic Web Conference*, pp. 260–276, Springer, 2013.

[18] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[19] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[20] "Filecoin: A decentralized storage network." https://filecoin.io/filecoin.pdf, 2018. Accessed: October 2018.

[21] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A taxonomy of blockchain-based systems for architecture design," in *Software Architecture (ICSA), 2017 IEEE International Conference on*, pp. 243–252, IEEE, 2017.

[22] V. Buterin, "Introduction to cryptoeconomics," 2017.

[23] V. Buterin, "Scaling adjudication with prediction markets." https://ethresear.ch/t/list-of-primitives-useful-for-using-cryptoeconomics-driven-internet-social-media-appl 3198, 2018. Accessed: November 2018.

[24] M. Goldin, "Token-curated registries 1.0." https://docs.google.com/document/d/1BWWC__-Kmso9b7yCI_R7ysoGFIT9D_sfjH3axQsmB6E, 2018. Accessed: November 2018.

[25] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds, "Efficient rdf storage and retrieval in jena2," in *Proceedings of the First International Conference on Semantic Web and Databases*, pp. 120–139, Citeseer, 2003.

[26] J. Broekstra, A. Kampman, and F. Van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *International semantic web conference*, pp. 54–68, Springer, 2002.

[27] D. C. Faye, O. Curé, and G. Blin, "A survey of rdf storage approaches," *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, vol. 15, pp. 11–35, 2012.

[28] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[29] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," in *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, vol. 2, pp. 205–213, IEEE, 2005.

[30] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM computing surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[31] L. Napster, "Napster," *URL: http://www. napster. com*, 2001.

[32] G. P. Specification, "vo. 4," *Available from World Wide Web: http://www9. limewire. com/developer/gnutella protocol 0.4. pdf*, vol. 155, 2003.

[33] K. F. S. Network, "Kazaa," 2002.

[34] M. Waldman, A. D. Rubin, and L. F. Cranor, "Publius: A robust, tamper-evident censorship-resistant web publishing system," in *9th USENIX Security Symposium*, pp. 59–72, 2000.

[35] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing privacy enhancing technologies*, pp. 46–66, Springer, 2001.

[36] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, *et al.*, "Oceanstore: An architecture for global-scale persistent storage," in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 190–201, ACM, 2000.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[38] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*, vol. 31. ACM, 2001.

[39] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 329–350, Springer, 2001.

[40] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*, pp. 53–65, Springer, 2002.

[41] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys & Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[42] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based routing," in *Parallel and Distributed Systems, 2007 International Conference on*, pp. 1–8, IEEE, 2007.

[43] M. J. Freedman, E. Freudenthal, and D. Mazieres, "Democratizing content publication with coral.," in *NSDI*, vol. 4, pp. 18–18, 2004.

[44] B. Cohen, "The bittorrent protocol specification," 2008.

[45] P. Haase, J. Broekstra, M. Ehrig, M. Menken, P. Mika, M. Olko, M. Plechawski, P. Pyszlak, B. Schnizler, R. Siebes, *et al.*, "Bibster–a semantics-based bibliographic peer-to-peer system," in *International Semantic Web Conference*, pp. 122–136, Springer, 2004.

[46] J. Zhou, W. Hall, and D. De Roure, "Building a distributed infrastructure for scalable triple stores," *Journal of Computer Science and Technology*, vol. 24, no. 3, pp. 447–462, 2009.

[47] I. Filali, F. Bongiovanni, F. Huet, and F. Baude, "A survey of structured p2p systems for rdf data storage and retrieval," in *Transactions on large-scale data-and knowledge-centered systems III*, pp. 20–55, Springer, 2011.

[48] M. Cai and M. Frank, "Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network," in *Proceedings of the 13th international conference on World Wide Web*, pp. 650–657, ACM, 2004.

[49] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. Van Pelt, "Gridvine: Building internet-scale semantic overlay networks," in *International semantic web conference*, pp. 107–121, Springer, 2004.

[50] E. Liarou, S. Idreos, and M. Koubarakis, "Evaluating conjunctive triple pattern queries over large structured overlay networks," in *International Semantic Web Conference*, pp. 399–413, Springer, 2006.

[51] M.-A. Sicilia, S. Sánchez-Alonso, and E. García-Barriocanal, "Sharing linked open data over peer-to-peer distributed file systems: the case of ipfs," in *Research Conference on Metadata and Semantics Research*, pp. 3–14, Springer, 2016.

[52] M. English, S. Auer, and J. Domingue, "Block chain technologies & the semantic web: a framework for symbiotic development," in *Computer Science Conference for University of Bonn Students, J. Lehmann, H. Thakkar, L. Halilaj, and R. Asmat, Eds*, pp. 47–61, 2016.

[53] P. N. Mendes, H. Mühleisen, and C. Bizer, "Sieve: linked data quality assessment and fusion," in *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pp. 116–123, ACM, 2012.

[54] A. Schultz, A. Matteini, R. Isele, C. Bizer, and C. Becker, "Ldif-linked data integration framework," in *Proceedings of the Second International Conference on Consuming Linked Data-Volume 782*, pp. 125–130, CEUR-WS. org, 2011.

[55] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri, "Test-driven evaluation of linked data quality," in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, (New York, NY, USA), pp. 747–758, ACM, 2014.

[56] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*, pp. 251–260, Springer, 2002.

[57] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[58] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy (SP)*, pp. 459–474, IEEE, 2014.

[59] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of rdf data management systems," in *International Semantic Web Conference*, pp. 197–212, Springer, 2014.

[60] "Ethereum name service." https://ens.domains/, 2018. Accessed: October 2018.

[61] J. Starr, E. Castro, M. Crosas, M. Dumontier, R. R. Downs, R. Duerr, L. L. Haak, M. Haendel, I. Herman, S. Hodson, J. Hourclé, J. E. Kratz, J. Lin, L. H. Nielsen, A. Nurnberger, S. Proell, A. Rauber, S. Sacchi, A. Smith, M. Taylor, and T. Clark, "Achieving human and machine accessibility of cited data in scholarly publications," *PeerJ Computer Science*, vol. 1, p. e1, May 2015.

[62] B. Fisch, "Poreps: Proofs of space on useful data." Cryptology ePrint Archive, Report 2018/678, 2018. https://eprint.iacr.org/2018/678.