INFORMATICA, 2022, Vol. 33, No. 2, 343–364

DOI: https://doi.org/10.15388/22-INFOR490

Blockchain-Based Transaction Manager for Ontology Databases

Timotej KNEZ*, Domen GAŠPERLIN, Marko BAJEC, Slavko ŽITNIK

University of Ljubljana, Faculty of Computer and Information Science, Slovenia
e-mail: timotej.knez@fri.uni-lj.si, domen.gasperlin@gmail.com, marko.bajec@fri.uni-lj.si, slavko.zitnik@fri.uni-lj.si

Received: January 2022; accepted: June 2022

Abstract. Knowledge graphs are commonly represented by ontology-based databases. Tracking the provenance of ontological changes and ensuring ontology consistency is important. In this work, we propose a transaction manager for ontology-based database manipulation that combines blockchain and Semantic Web technologies. The latter is used for the efficient querying and modification of data, whereas the blockchain is used for the secure storage and tracking of changes. The blockchain enables a decentralized setup and data restoration. We evaluate our solution by measuring cost and time. Our solution introduces some overhead for updates whereas querying works at the same speed as the underlying ontology database.

Key words: Semantic Web, blockchain, ontologies, databases, IPFS, Ethereum.

1. Introduction

The World Wide Web is one of the most commonly used platforms for computer applications today. While it contains a large amount of knowledge on a wide variety of topics, most of that knowledge is captured in unstructured text documents. These documents are designed to be understood by humans and are difficult to use for computer systems. The inventor of the World Wide Web, Tim Berners-Lee, famously claimed that the information presented on the Web should be accessible as raw data presented in a well-defined structured format. The data would thus be more useful for other applications. He described the concept of the Semantic Web, where in addition to human-readable formats, the information would also be available in a structured form designed to be used by computers. In order to help reach this goal, the World Wide Web Consortium (W3C) defined a set of technologies that are recommended when developing the Semantic Web. For modelling human knowledge in a way that is useful for computers, ontology representations are commonly used. In computer science, ontologies are collections of entities, their properties, and connections between them. Building and updating ontologies present many challenges. One of the main challenges is storing an ontology in such a way that anyone could use it and

*Corresponding author.

contribute new knowledge. This could be achieved by setting up centralized servers. That, however, would require someone constantly maintaining and paying for the infrastructure. Once the maintenance stopped, the ontology would also stop being available and would be lost.

In recent years, we have seen a rise in the popularity of blockchain technologies. Public blockchains like Bitcoin (Nakamoto, 2009) work using a peer-to-peer network and a set of advanced cryptographic algorithms to store a list of transactions between its users. This way they can provide transparency, decentralization, and tamper-resistance, which are all features that would also be useful for managing public ontologies. There are different varieties of blockchain networks. On public blockchains, anyone can read and use the stored data. There are also private blockchains that limit who can see and change their data. Since a private blockchain allows us to ensure that only trusted users get access, there is less need for strict security protocols. Some blockchains use some concepts from public and some from private blockchains. Those chains are called hybrid blockchains. The idea of using blockchains for storing data was further expanded in 2015 when the Ethereum network (Buterin, 2014) enabled many other applications to take advantage of the technology using smart contracts. Smart contracts are programs that we can publish on the Ethereum blockchain that ensure the execution of agreements described in them. In our project, we use smart contracts to store the data about each change on the Ethereum blockchain.

Blockchains ensure decentralization, security, and transparency. While blockchains provide secure and decentralized storage, they are not designed to store large amounts of data. This limitation comes from the fact that every client has to download all of the data that is stored on the blockchain. In order to store larger amounts of data in a decentralized way, we propose the use of the IPFS network (Benet, 2014). IPFS (interplanetary file system) is a P2P system that distributes storage among multiple computers in a network. The network is designed for storing larger files. We can access the files stored in the network using their content identifiers. Each identifier is comprised of multiple parts: the encoding prefix, version, content type code, and the hash of the document. The identifiers are short enough to be stored on the blockchain using a smart contract. This way we can combine the two technologies to store larger files while keeping the security and transparency that the blockchain provides.

In our work, we combine the blockchain with the technologies of the Semantic Web to produce a system for managing ontologies. Our system uses the Ethereum smart contracts and the IPFS network to store an ontology in a completely decentralized way. This allows us to distribute a public ontology to its users without needing centralized servers. We can also enable users to submit their own changes to the ontology. All the changes to the ontology are tracked, so anyone can revert the ontology back to a previous state and continue developing it as their own branch. The implementation and evaluation of the proposed system is publicly available in the source code repository (Gašperlin, 2021). The main contributions of our work are as follows:

a) Implementation of the proposed solution into an ontology-based database transaction manager would provide strong security via decentralization and authorized change logging.

b) Based on existing proposed systems, the proposed system allows for complete data loss protection and complete history of changes that enable restoration to any point in history.

The rest of the paper is organized as follows: in Section 2 we present previous work related to the Semantic Web, blockchain technologies, and combining the two. In Section 3, we present our transaction manager for ontology databases that uses Ethereum and IPFS to track changes made to an ontology. In Section 4, we present data flow and describe common usage patterns. After that, in Section 5, we run a series of tests to determine the performance and cost of using our transaction manager. Finally, in Section 6, we comment on the results and propose some directions for future work.

2. Background and Related Work

In this section, we introduce previous work that has explored the use of ontologies and other Semantic Web technologies. We also present work related to blockchain technologies and a combination of the two.

2.1. *Semantic Web*

The idea of a Semantic Web was proposed by Tim Berners-Lee at the very first World Wide Web Conference in 1994 (Shadbolt *et al.*, 2006). Many years later, however, it has still not been fully realized. The technology stack for implementation of the Semantic Web was later standardized by the W3C organization. The semantic Web is designed to be an extension of the World Wide Web, where the meaning of information is well defined. The main focus is put on the data instead of the documents. On the Semantic Web, data is presented by entities and the relations between them.

The basis for a Semantic Web are unique identifiers of resources that can be used to reference and differentiate between the resources. For that purpose, we use uniform resource identifiers (URI). The most common type of a URI is the URL that defines the protocol and the location on the computer network required for accessing a resource.

In addition to resource identifiers, another technology defined by the W3C is the data model for representing information. W3C standardized the use of the Resource Description Framework (RDF) model, described by Decker *et al.* (2000), for data representation on the Semantic Web. The RDF model uses simple statements to describe resources and connections between them. When working with ontologies, it is useful to have some unified sets of relations that can be used to describe entities. For this purpose, the RDFS and the OWL languages were created to expand the RDF and provide better descriptiveness (McBride, 2004; Horrocks *et al.*, 2003; McGuinness and Van Harmelen, 2004).

The Semantic technology stack also defines the SPARQL language described by Pérez *et al.* (2006). SPARQL is a language for querying an ontology. It supports a set of statements for retrieving data from an ontology. It also supports commands for inserting and deleting relations captured in an ontology. The specification also defines some other standards that are not relevant to our proposed system.

In this work, we focus mostly on the part of the Semantic Web related to building ontologies. The ontologies are considered to be the backbone of the Semantic Web (Taye, 2010). In the field of computer science, ontology is a model of entities and the relations between them (Stevens *et al.*, 2000; Noy and McGuinness, 2001). Ontologies have become one of the most common representations for storing general human knowledge in computer systems (Noy and McGuinness, 2001). Over the last years, many different ontologies have been built with the aim of capturing as much Semantic knowledge as possible (Lehmann *et al.*, 2015; Bollacker *et al.*, 2008; Speer and Havasi, 2012; Shortliffe, 1976; Donnelly, 2006). One of the best-known ontologies is DBpedia (Lehmann *et al.*, 2015), which contains general knowledge gathered from infoboxes on Wikipedia. We decided to demonstrate our system on the DBpedia ontology. Ontologies can also contain more specific domain knowledge. For instance, the SNOMED CT (Donnelly, 2006) ontology and the MYCIN (Shortliffe, 1976) ontology contain medical information designed to help doctors with their work.

2.2. *Blockchain Technologies*

The rise of blockchain technology started with the cryptocurrency Bitcoin in 2009 (Nakamoto, 2009) when a peer-to-peer electronic cash system was presented to allow online payments to be sent to people without the need for centralized intermediaries. This has also solved the double-spending problem. However, the idea of a chain of blocks secured by cryptography was introduced by Stuart Haber for timestamping digital documents in 1991 (Haber and Stornetta, 1991).

Blockchain technology has been evolving through generations. First-generation (Bitcoin, Litecoin, Dogecoin) was focused on creating a digital currency. The second-generation (Ethereum Classic) extended the concept to the use of smart contracts. They enable more advanced use cases besides the transfer of currency to support the digital economy. Smart contracts follow the terms of the agreement. An example of such platform is Ethereum but in the early stages, such smart contracts were error-prone and contained security vulnerabilities. Ethereum co-founder Gavin Wood even coined the term *web3* as a decentralized online ecosystem based on blockchain. It could be the future of the world wide web (Edelman, 2021). The third generation (Cardano, Polkadot) created blockchain applications. With them, we can create a digital society where people could participate in broader economic activity in the areas such as health, art, education, and culture (Efanov and Roschin, 2018). Based on accessibility we can differentiate between consortium, private and public blockchains. In public blockchains, anyone can join the network and participate in it. In private blockchains, only a selected few participants can be part of the network. Consortium blockchains have properties of both public and private networks. For consortium blockchains, permissions can be set as needed. They can be used to connect people from multiple organizations.

With smart contracts on Ethereum other cryptocurrencies or tokens can be created. Fungible tokens like Ether can be exchanged and traded as they are identical. On the other hand, we have non-fungible tokens. Each such token is unique and is not identical to other

tokens. They can be used to represent rights, identities, and other real-world items such as art. In our application, we only use the ether tokens, which are fungible to pay for smart contract operations. The use of custom tokens is interesting for future expansion, as they could be used for the purpose of managing user permissions or as a payment for certain operations on the ontology.

2.3. *Semantic Web and Blockchain Integrations*

The combination of Semantic Web and blockchain technologies has seen some attention in the literature recently. One project combining the two technologies was presented by Hector and Boris (2020). They created an interface in the form of an ontology called BLONDiE, that uses a Semantic representation to convey the current information about a blockchain. Multiple projects also explored ways of using blockchains to make the Semantic Web better. One such work is presented by Ugarte (2017), who presents an idea about a futuristic Internet based on blockchains. English *et al.* (2016) present the potential benefits of combining Semantic technologies with a blockchain. The work of English *et al.* is later expanded by Cano-Benito *et al.* (2019). Both papers focus on using blockchain technologies in the Semantic Web definitions themselves. By doing this the Semantic Web can be improved. They also present some scenarios in which the blockchain could be improved by using Semantic Web technologies. On the other hand, our aim is not to replace parts of the Semantic Web with blockchain, but rather to use the blockchain to enable the construction of a communal ontology without relying on a centralized governing entity.

Naim and Klas (2019) present a Semantic blockchain framework, that allows storing data on a blockchain in the form of an ontology. This helps with the efficient retrieval of data from the blockchain. Their approach requires everyone connected to the blockchain network to store the entire ontology. This raises scalability concerns for use cases that require storing large ontologies.

Another solution using blockchain technologies is presented by Markovic *et al.* (2019). They implement a system for tracking food transport and determining if each part was compliant with the appropriate legislation. Data is stored in the Hyperledger Fabric blockchain that allows buyers to check if the transport process was appropriate.

Our application focuses on using a blockchain for tracking changes in an ontology. Tracking ontology changes is not a new concept. We present the existing tools that are related to our use case in Table 1. Some tools for tracking changes for the purposes of undo and redo functionalities are already present in the Protégé editor for ontologies (Liang *et al.*, 2005). These tools are very limited since all changes are lost after a program restart. An improvement for change tracking in Protégé was proposed by Khattak *et al.* (2009). They created a plugin for the Protege editor that tracks every change that the user makes and allows a rollback to any previous version. Their solution keeps the changes stored in one place, which enables the tracking of each change from the beginning of an ontology. In their solution, all changes are only stored on a single computer and are available to only one user. We aim to use distributed storage technologies to store the changes in a decentralized and distributed way that protects us from data loss and enables collaboration between multiple users.

Table 1
Comparison of projects that are the most similar to ours. All of the systems using a blockchain platform use Ethereum in combination with IPFS.

Project	Goal	Collaborative editing	Blockchain platform	Off-chain solution	File history	Persistent storage	Data loss protection	Semantic checks
Protégé history	Tracking ontology changes for undo and redo operations	✗	✗	✓	✓	✗	✗	✗
Nizamuddin *et al.* (2019)	Tracking changes to documents	✓	✓	✗	✓	✓	✓	✗
Khattak *et al.* (2009)	Tracking changes to the ontology	✗	✗	✓	✓	✓	✗	✓
Jianjun *et al.* (2020)	Sharing documents only with trusted users	✓	✓	✗	✗	✓	✓	✗
Steichen *et al.* (2018)	Add access control using blockchain to the IPFS network	✓	✓	✗	✗	✓	✓	✗
Our system	Track changes to the ontology on a blockchain	✓	✓	✗	✓	✓	✓	✓

We propose a solution for ontology management, that uses the Ethereum blockchain combined with IPFS to track changes. This concept is somewhat similar to the work done by Nizamuddin *et al.* (2019). They used Ethereum and IPFS to keep track of changes when editing documents. Their application works in a similar way to version control software. They also implemented the limitation that a change is only applied if it is approved by at least two-thirds of the approvers. The main difference between our system and the one proposed by Nizamuddin *et al.* is that their system is designed to work with general documents without understanding their structure, while ours is designed as a transaction manager for an ontology database. Our system uses the Apache Jena framework to parse and execute the queries written by a user. Thus it can perform a variety of checks for the correctness of a change, like checking its syntax, checking the ontology consistency after the execution of a query etc. Jianjun *et al.* (2020) also combined IPFS with Ethereum to create a system for sharing documents that can only be viewed by authorized users. This is achieved by using an Ethereum smart contract for checking the identities of the users. An approach similar to Jianjun *et al.* was also proposed by Steichen *et al.* (2018). They use Ethereum and IPFS for access control. This way they solve the problem of only allowing some users to access documents stored on the IPFS network.

A large majority of the existing solutions for integrating semantic web with blockchain technologies use the Ethereum blockchain. The main reason for that is that they rely on the use of Ethereum smart contracts. The Ethereum smart contracts can also be used on any other network compatible with the Ethereum virtual machines. Because of that, there are several alternatives to using Ethereum. We found that the most perspective alternatives are the BNB chain (Binance team, 2022) and the Fantom platform (Choi *et al.*, 2018). An advantage of using Ethereum is that it has the most users making it very secure, while the alternatives offer cheaper prices for executing smart contract functions.

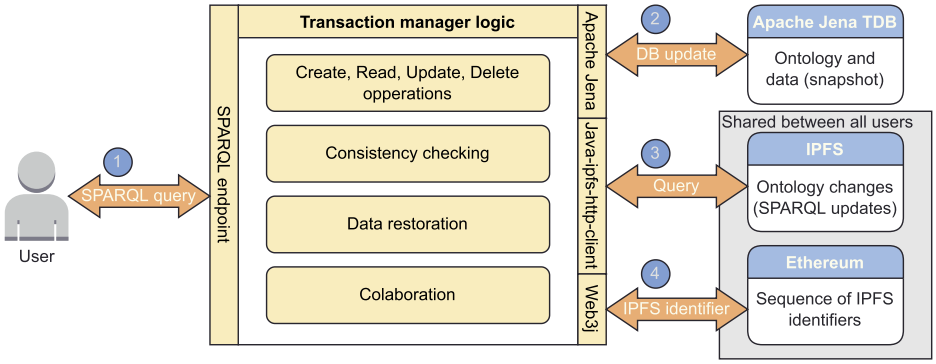

Fig. 1. Architecture of the proposed system. The user updates the database using a SPARQL query. (1) The user sends a query to the SPARQL endpoint. (2) Our transaction manager uses Apache Jena to execute the update on the local ontology database. (3) Transaction manager uses the java-ipfs-http-client to upload the query to IPFS. (4) Transaction manager uses the Web3j library to store the IPFS identifier to the Ethereum smart contract.

3. Semantic Data Empowered by Blockchain

We present an architecture for a system combining Semantic Web and the blockchain technologies (Fig. 1). The system tracks changes to the ontology on a blockchain. The implementation of the system is available in a public source code repository (Gašperlin, 2021). By tracking ontology changes on a distributed blockchain, our solution ensures safe storage of the ontology and also automates checking for its consistency. Among the main advantages of using a blockchain-based ontology management system are: (a) the ability for anyone to get the latest version of the ontology and contribute to it, (b) no need for dedicated ontology storage, (c) the ability to rollback to any stage of the build and (d) immutability of change history.

Our proposed system allows the user to interface with the ontology using the SPARQL language (Pérez *et al.*, 2006). This allows the user to query and edit its content as well as create and delete records. Whenever a user executes a query that changes the ontology, the change is also automatically stored on the blockchain. This allows us to reconstruct the entire ontology from the blockchain in the event of a complete data loss. The use of a public blockchain also means that anyone wanting to contribute or use our ontology can reconstruct the ontology from the blockchain. After that, they can use the same system to apply their own changes and share them on the blockchain. The use of cryptographic methods also means that the identity of the author of each change can be tracked securely. By checking the consistency of the ontology before applying a change to the blockchain we can also ensure that no contradictions are introduced into the ontology.

The architecture is shown in more detail in Fig. 1. The user interfaces with the application logic using the SPARQL language. The current ontology is stored locally in the Apache Jena Triple Database (TDB). The application uses the Apache Jena library (Siemer, 2019) to access the Apache Jena TDB. The Apache Jena library was selected for the implementation as it supports RDF databases, SPARQL querying, and reasoning for checking the ontology consistency. It lacks support for the OWL 2 language,

but if that is a problem, we recommend switching to the owlapi library. Each change is also recorded on the blockchain. Since storing large documents on a blockchain itself is not efficient and would be very expensive, we store the data about the change on the IPFS network. The application then stores the content identifier of the file stored on the IPFS network to the Ethereum blockchain. The application uses the java-ipfs-http-client library (Preston *et al.*, 2020) for communication with the IPFS network and the Web3j library[1] to communicate with the Ethereum network and for the creation of smart contracts. All libraries are managed using Maven. This selection of tools is just an example of tools to validate our theoretical proposal of creating an *ontology-based transaction manager, empowered by blockchain*. The architecture of the system is presented in Fig. 1.

When using our ontology management system the user should not bypass our system by editing the local ontology directly. This could result in a difference between the state of the ontology recorded on the blockchain and the local ontology. In this case, a consistent state could be achieved once again by restoring the local ontology from the blockchain.

Before we can run the program we have to define a set of parameters that the program requires. Parameters can be set in the configuration file of the program. We have to set the following parameters:

a) Path to the file containing SPARQL commands to execute;
b) IPFS node for connecting to the IPFS network;
c) Ethereum node for connecting to the Ethereum network;
d) A user account with funding for the smart contract (Private key or a wallet address with a password).

We publish changes by first uploading them to the IPFS network. As a result, we get an identifier that we can later use to retrieve the changes file. The identifier gets stored on the Ethereum blockchain using a smart contract that gets created at the first launch of our program. When one user creates a change, others can use the smart contract and the IPFS network to get information about the change and update their local ontologies. The smart contract is written in the Solidity language and translated to a Java class. It is not storing the entire queries that might be large in size, but only the content identifiers that can be used to retrieve the full queries from the IPFS network.

The smart contract contains 6 methods that get called by the transaction manager (see Algorithm 1). The initial ontology refers to a set (or multiple sets) of triplets that get inserted when the ontology is initialized. The migration refers to a SPARQL query that has been executed on the ontology to change its data.

3.1. *Selection of an Appropriate Blockchain Platform*

We decided to use the Ethereum blockchain for storing the content identifiers of each change to the ontology. The reason for using the Ethereum network is that the Ethereum smart contracts are also supported by various different blockchain systems. We demonstrate our system on the Ethereum network which is the most well-known and offers high

[1] Web3j Github: https://github.com/Web3j/Web3j

Algorithm 1 Ethereum smart contract

```
1:  procedure ADD INITIAL ONTOLOGY (IPFS IDENTIFIER)
2:      initialOntologies.push(IPFS identifier)
3:  procedure GET THE NUMBER OF INITIAL ONTOLOGIES return initialOntologies.length
4:  procedure GET INITIAL ONTOLOGY (INDEX)
5:      if index < initialOntologies.length then return initialOntologies[index]
6:      else
          return ""
7:  procedure ADD MIGRATION (IPFS IDENTIFIER)
8:      migrations.push(IPFS identifier)
9:  procedure GET THE NUMBER OF INITIAL ONTOLOGIES return migrations.length
10: procedure GET MIGRATION (INDEX)
11:     if index < migrations.length then return migrations[index]
12:     else
          return ""
```

security due to a large number of miners. One of the drawbacks of using Ethereum for production systems is the high cost of smart contract transactions. To avoid that, the user could easily reconfigure our system to work with a different blockchain that also supports Ethereum smart contracts.

3.2. *Consistency Checking*

The idea behind consistency checking is that we can apply some rules to the relations in an ontology. For instance, if we have a relation stating that a person was born on a certain day, we can decide not to allow another relation stating that the same person was born on another date. In order to prevent such cases, we need a method for checking consistency based on some reasoning rules. Apache Jena supports multiple inference engines (reasoners) for deriving assertions from base RDF graph and for checking its consistency (Apache Jena, 2021). The mechanisms differ in their capabilities and complexities:

Transitive reasoner is a simple reasoner that provides support for checking the transitive and reflexive properties of the *subPropertyOf* and *subClassOf* relations.

RDFS reasoner provides an implementation for the RDFS entailments. The user can configure which entailments to use.

OWL, OWL mini, and OWL Micro reasoners implement a part of the OWL/Lite language.

Generic rule reasoner provides support for user-defined rules. For example, the rule could automatically assign rdf:type BigCity to the cities that have a population greater than 1 million.

We only publish the changes to the blockchain if the resulting ontology is consistent. One limitation of our approach for consistency checking is that the checks are done locally

before a change is applied. This means that a malicious user could bypass the checks and apply a contradictory change directly to the blockchain. This can be mitigated by checking consistency when fetching the changes and reverting transactions that cause invalid states. In order to completely resolve this issue in the future, we propose the use of a voting-like mechanism where a change has to be confirmed by some number of users in order to become a part of the ontology. This mechanism can be implemented similarly to the voting mechanism proposed by Nizamuddin *et al.* (2019). In this case, we would have a group of trusted users, who have the ability to approve transactions. Once a transaction has been approved by two-thirds of the trusted users, it would become part of the ontology. In our case, the process of approving a transaction would be done automatically by the reviewers' transaction managers. In order for someone to become a trusted user, they would also need the approval of a large enough percentage of existing trusted users. This way the changes that cause an inconsistent state would not get included in the shared ontology. This mechanism is not a part of our current implementation but could be implemented in the future.

4. Common Usage Patterns

In this section, we present some of the common usage patterns that are supported by our application. Each pattern describes steps that are taken in the application when completing a task. We determine that the most important scenarios for our system are (a) the initialization of an ontology, (b) running a query, (c) recovering an ontology from the blockchain and (d) collaborative development of an ontology.

4.1. *General System Data Flow*

When the transaction manager gets executed, the system first checks if the local ontology is present. If there is no local ontology, it either reconstructs the ontology from the blockchain (see Section 4.4) if it is available or initializes a new ontology (see Section 4.2). Once the local ontology is present, the system checks for new changes that might have been submitted to the blockchain by other users and applies them to the local ontology (see Section 4.5). After that the queries provided by the user get executed on the local ontology and written to the blockchain (see Section 4.3).

We present the interactions between each part of the system that take place when a user submits a query to an already initialized ontology (Fig. 2). When a query is sent to our transaction manager, it first checks for new changes using the Ethereum smart contract. If any new changes have been committed since the last synchronization, it loads their identifiers. The identifiers allow the transaction manager to locate the details about each change on the IPFS network. The transaction manager then uses the identifiers to retrieve the SPARQL queries that have been executed during each of the changes and runs them on the local ontology database. After that, the state of the local ontology is synchronized with the shared state.

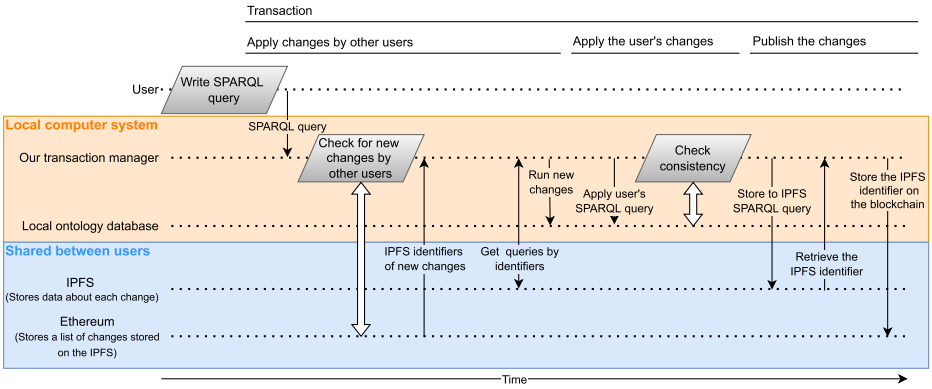

Fig. 2. A typical data flow between components of our system when a user runs a query using our transaction manager. In the presented scenario, the system is already initialized and the user is running one or more SPARQL queries. The first part of the process is responsible for retrieving new changes submitted by other users of the ontology. The second part is responsible for applying and publishing the changes done by this user.

Once the local ontology is in a synchronized state, the user's query gets executed and the transaction manager checks the ontology for consistency with all of the reasoning rules. If the ontology is consistent, the query gets written to the IPFS network. After that, the transaction manager writes the IPFS content identifier of the query to the Ethereum blockchain using the smart contract. By storing the queries on the IPFS network and only storing their content identifiers on the blockchain we avoid the problems and cost of storing large amounts of data on the blockchain. This is important since some queries for adding ontology records can be very large.

4.2. *Initializing the Ontology*

The first use case that we describe is responsible for the initial loading of an existing ontology into our system. In this case, we start with a pre-existing ontology stored locally as a set of files containing the ontology triplets. The local ontology of the system is not yet set up and the ontology is not yet stored on the distributed storage. During the process, the ontology gets loaded into our local database and stored on the IPFS and blockchain. This process is important since it allows us to use our system on an already existing ontology. Our system also supports initialization from an empty ontology or an ontology stored on the IPFS and blockchain. The latter is presented in more detail in Section 4.4. The process of initialization using a local ontology has the following steps (presented in Fig. 3):

a) The ontology data is first loaded into the local Apache Jena database.
b) We use the Apache Jena reasoner to check the loaded ontology for consistency.
c) If the ontology is consistent, it is uploaded to the IPFS network and its content identifiers are saved to the Ethereum smart contract.

After the initialization, we can track the changes of the ontology. We can also make some guarantees about the integrity of the tracked data. Ontology data at the content

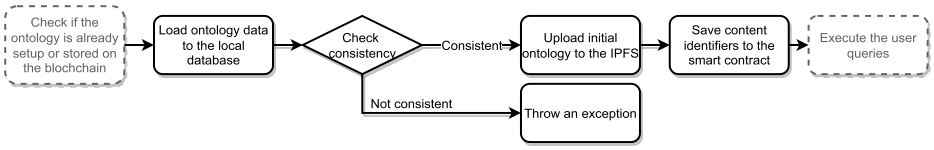

Fig. 3. Initializing the ontology management system from an existing ontology stored locally. In this use case the parts of the process shown in gray are not relevant as the ontology is not yet initialized and there are no user queries to execute.

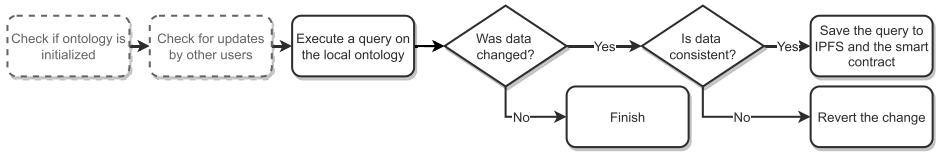

Fig. 4. The process of executing a query using our system. The parts of the process shown in gray are not relevant for this use case as the ontology was already initialized and there are no changes submitted by other users.

identifier has not been altered. This could not be guaranteed if we were using location addresses. Journal of ontology changes is kept, which proves the time and author of any change. The authors of each change to the ontology are identified under the Ethereum pseudo-anonymous address.

4.3. *Running a Query*

The second scenario is querying the data in the ontology (presented in Fig. 4). Queries are created in SPARQL language which allows both reading and updating the ontology. If the query only performs a read operation, the system only returns the local data. This way there is no need for the usage of the Ethereum or the IPFS which might slow down the performance. In the event that the query changes, creates, or deletes entries in the ontology, all of the changes are also checked for consistency and written to the Ethereum blockchain. Writing to the blockchain is done in two parts. First, the query is stored on the IPFS and the system generates an IPFS content identifier. Next, the identifier is stored in the smart contract on the Ethereum network. Running a query follows these steps:

1. Execute the query on the local ontology;
2. If the query changes the data, check for consistency, otherwise finish;
3. If the data is consistent, commit the change, otherwise, throw an exception;
4. Store the change data to the IPFS;
5. Save the IPFS identifier to the Ethereum blockchain.

After the ontology data is updated, other users can download the changes from the blockchain and apply them to their local ontologies. This way each change can be shared between everyone that is working on the ontology. The collaboration scenario is explained in more detail in Section 4.5.

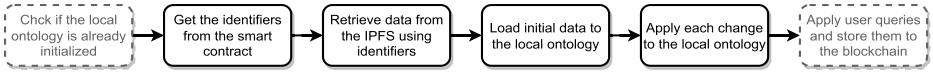

Fig. 5. The process of restoring the ontology from the blockchain. The steps presented in gray are not relevant for this use case as the local ontology was not yet initialized and there are no queries submitted by the user.

4.4. *Recovering from the Blockchain*

The use of the blockchain enables full recovery of the ontology. This means that if any of the data in the local ontology is lost, it can be restored from the blockchain. The same process can be used when a new user wants to get the shared ontology. The recovery process is presented in Fig. 5. The recovery process works in a similar way to the initialization of the ontology except that the ontology is loaded from the Ethereum network instead of the local file system. Loading the ontology is done in two parts. First, all of the identifiers for the initial data and every recorded change are loaded from the smart contract on the Ethereum network. Next, the actual initialization data and queries are downloaded from the IPFS network using the retrieved identifiers. The steps for the ontology recovery are the following:

1. Migration identifiers are retrieved from the blockchain using the smart contract;
2. Information about changes is loaded from the IPFS using the identifiers;
3. Initial data is loaded to the local ontology;
4. All of the changes are applied to the local ontology.

A similar process can be also used for updating an existing local ontology to the latest version. Our implementation stores the queries that have been already executed, so it can automatically skip the changes from the Ethereum that have been applied already.

4.5. *Collaborative Ontology Building*

As mentioned in the previous scenario, we can use the data stored on the blockchain in order to enable collaboration between multiple people when working with the ontology. Once the ontology exists on the blockchain, new contributors can use the recovery process described in Section 4.4, to recreate the entire ontology locally. After that, they can apply their own changes to the ontology. The changes get automatically stored on the blockchain, so the other users can download them and apply them to their local ontologies. The process is shown in Fig. 6. The collaboration between person A and person B would follow these steps:

1. A initializes the ontology and applies some changes;
2. B uses the recovery procedure to get the latest version of the ontology;
3. B applies changes to the ontology;
4. A downloads the changes from the blockchain and applies them to the local ontology.

The option for the collaborative building of the ontology enables the creation of a completely open ontology, where anyone could contribute their knowledge. Since this

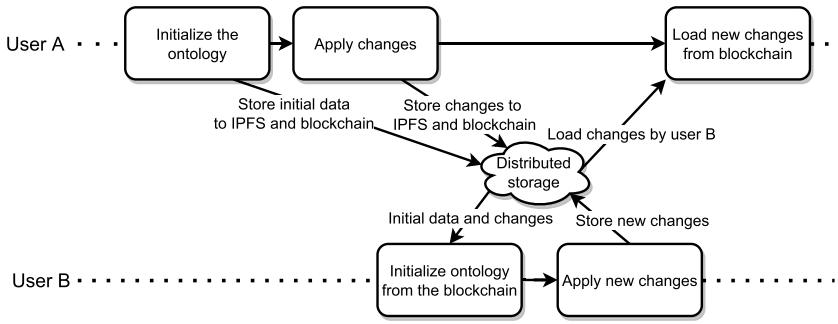

Fig. 6. Collaborative building and editing of the ontology using the Ethereum blockchain for synchronization.

ontology can be stored on a decentralized blockchain, there is no need for centralized servers and their maintenance.

5. Evaluation

An important aspect of the proposed solution is its performance and operational cost. This is especially important, since using smart contracts tends to be slow and can also get expensive quickly. Each operation on an Ethereum smart contract requires a certain amount of gas to perform. Gas is a currency that we use in order to pay for the processing and storage required to execute an operation. We performed a series of tests to estimate the performance and the cost of using our solution. Each test was performed ten times in order to compute the mean values and standard deviations.

5.1. *Evaluation Setup*

All tests were executed on a Mac computer with 16 GB of memory and an Intel 8-core processor. During our tests, the heap size of our program was set to 2 GB. We tested the performance of the most common use cases by running a scenario 10 times. The scenario consisted of three parts, performed by two different users and contained the initialization of the ontology, its restoration from the blockchain, execution of all query types, and collaboration between the user. This way we covered all of the presented use cases. We measured the time that each operation took to execute as well as the amount of gas that has been required for the Ethereum operations. We computed the price of the required gas in Euro based on the price at the evaluation time (22.12.2021). In Tables 2, 3, 4, 5, and 6 we present the average values as well as standard deviations for times and prices of each operation. For testing, we used a public Ethereum test network called Rinkeby[2]. We believe that our results are comparable to using the main Ethereum network; however, by using Rinkeby we do not have to pay for the smart contract transactions. We also used the

[2]Rinkeby Website: https://www.rinkeby.io/

public IPFS network with one node running on the local computer. After each run of the evaluation, we have used the IPFS garbage collector to clear the cache of the local IPFS node. By doing so, we have ensured that the cached data from the previous runs had no effect on the next ones.

In our tests, we evaluated the effect of consistency checking separately from the main tests. That is because we want our main tests to reflect the effects that the use of blockchain has on the performance of the system working with large ontologies. Our tests have shown that consistency checking in Apache Jena becomes memory intensive when working with large ontologies, which prevents us from running consistency checking on our entire ontology.

5.2. *Evaluation Ontology*

For testing the transaction manager, we needed a large ontology to demonstrate common use scenarios. By using a large ontology we can demonstrate the performance of our system for larger projects. There are several such ontologies that are publicly available. We decided to use the DBpedia (Lehmann *et al.*, 2015) since it is one of the most well-known general knowledge ontologies. We decided to only load the parts of DBPedia, necessary for running our test queries. The final size of the database was 2.8 GB. The parts that we used are: (1) the ontology structure, (2) home pages of resources, (3) numerical properties of entities, and (4) entity types. By including these parts of the ontology, our database contained all of the entities, their numeric properties, home pages, and types. The evaluation ontology, therefore, contained approximately 7 million entities of about 800 different types with around 3000 different properties. Approximately 6.5 million of the entities fall under one of the DBpedia types, while the rest have a type from another ontology. Altogether there were around 9 million triplets in our testing ontology.

5.3. *Initialization*

During the initialization, the smart contract was first published to the Ethereum network and then all four parts of the data from DBpedia were stored in the local database. The files required for the initialization of the ontology were uploaded to the IPFS network, so they can be retrieved by anyone using their identifiers. The IPFS identifiers were then stored on the blockchain using the smart contract. We compared this with the initialization process when only using the Apache Jena library without the blockchain. The times and gas cost of each operation is presented in Table 2. In this scenario, the steps for publishing the initial ontology to the IPFS and storing the IPFS identifiers in the smart contract were performed in four steps. In the table, we present the sum of all four times and prices. The reason for the four parts is that we used four parts of the DBpedia in our testing and each part is stored in a separate file.

Most of the time was spent loading the data to the Jena TDB database. This part has to be performed even if we are only using a local ontology without the blockchain. The reason for a slow operation is that our initial ontology contains a large amount of data,

Table 2
The time required and the cost of performing an initialization of the transaction management system with the DBpedia ontology. The columns under "Blockchain transaction manager" describe the performance of our system, while the last column describes only using local ontology without blockchain. The steps "Publishing initial data to IPFS" and "Publishing identifiers" were performed in four parts, since we have four initial files.

	Blockchain transaction manager			Apache Jena
Operation	Time [s]	Gas used	Price [€]	Time [s]
Loading initial data to Jena	311 ± 14	/	/	311 ± 14
Publishing the smart contract	17 ± 0.11	310149 ± 0.0	1.09 ± 0.0	/
Publishing initial data to IPFS	9.52 ± 0.17	/	/	/
Publishing identifiers	70 ± 10	395652 ± 0.0	1.39 ± 0.0	/
Total	$\mathbf{407 \pm 13}$	$\mathbf{705801 \pm 0.0}$	$\mathbf{2.47 \pm 0.0}$	$\mathbf{311 \pm 14}$

which requires a substantial amount of processing before it can be used. In addition to that, we have to perform blockchain synchronization steps in our implementation which take about two minutes.

5.4. *Running a Query*

The most common operation is running a query. The steps required to run a query are described in Section 4.3. We evaluated the performance of this scenario by running three queries. The first query only retrieved values and did not change the data, so the smart contract did not get updated. The second query performed the insert operation, and the third query deleted an entity in the ontology. The final two queries changed the ontology so they had to be stored on the IPFS and recorded in the smart contract. The time required to run each query is presented in Table 3. The select query (1) is performed locally and does not require a blockchain update. The insert query is first performed locally (2a), then the query is uploaded to the IPFS network (2b) and stored by an identifier. Finally, the identifier is stored on the blockchain using the smart contract (2c). The same process is used for the delete query (3a–3c).

The query for retrieving data took a similar amount of time compared to only running locally without the blockchain. This was expected since we only use the local database to retrieve the data. The two queries for inserting and deleting data both took significantly longer to complete on our implementation than on a system only using Apache Jena. This was also expected since changes to the ontology require the creation of a record on the blockchain and communication with the IPFS network. Most of the time required for these queries was used for updating the smart contract.

5.5. *Rebuilding the Ontology After a Data Loss*

One of the advantages of using blockchain is that we can recover the entire ontology in the event of a data loss. To showcase that, our third test was a complete recovery of the ontology. This is done in three steps. First, the IPFS identifiers are retrieved from the blockchain. Since this is only a read operation, it can be performed quickly and does not

Table 3
Time spent on each of the test queries for the implementation using Ethereum blockchain (columns under "Blockchain transaction manager") and a local Apache Jena instance with no blockchain (last column). We performed three queries (written in bold). Insert and delete queries require multiple steps so operations 2a–2c and 3a–3c show data for individual steps.

	Blockchain transaction manager			Apache Jena
Operation	Time [s]	Gas used	Price [€]	Time [s]
1: SELECT	1.28 ± 0.26	/	/	1.28 ± 0.26
2a: INSERT local	0.41 ± 0.03	/	/	0.41 ± 0.03
2b: INSERT upload to IPFS	0.2 ± 0.03	/	/	/
2c: INSERT save to Ethereum	17 ± 4.54	94677 ± 0.0	0.33 ± 0.0	/
2: INSERT total	18 ± 4.52	94677 ± 0.0	0.33 ± 0.0	0.41 ± 0.03
3a: DELETE local	0.42 ± 0.01	/	/	0.42 ± 0.01
3b: DELETE upload to IPFS	0.2 ± 0.02	/	/	/
3c: DELETE save to Ethereum	16 ± 0.15	111777 ± 0.0	0.39 ± 0.0	/
3: DELETE Total	16 ± 0.15	111777 ± 0.0	0.39 ± 0.0	0.42 ± 0.01

Table 4
Time and cost of each operation required for rebuilding an ontology from blockchain. Since all of the performed operations only read data, they do not require payment. The implementation without blockchain cannot perform this scenario.

Operation	Time [s]	Gas used	Price [€]
Read IPFS IDs from Ethereum	0.59 ± 0.02	/	/
Download initial ontology from IPFS	0.59 ± 0.02	/	/
Apply initial ontology to local DB	309 ± 9.87	/	/
Get update IDs from Ethereum	0.15 ± 0.0	/	/
Download update queries from IPFS	0.02 ± 0.01	/	/
Apply update queries	1.07 ± 0.04	/	/
Total	$\mathbf{311 \pm 9.88}$	/	/

require gas payment. After that, the data is downloaded from the IPFS network. Finally, the data is loaded to the Jena database. These three steps are repeated first for the initial data and second for the transactions that were applied on top of the initial database. Times for all parts of the operation are presented in Table 4. Just like in the initialization scenario, most of the time is spent loading the ontology to Jena. This is expected since the ontology is still the same size. This is also the reason that the entire process of restoring the ontology from the blockchain takes a similar amount of time compared to the loading of initial data in the initialization of the ontology in Section 5.3.

When recovering an ontology, we can also choose to only apply changes up to a certain point. This way we can get an ontology from a certain time instead of using the latest one. A user could also use our system to create a new branch of the ontology. Creating a branch requires the creation of a new Ethereum smart contract.

Table 5
Time and cost of each step in the collaborative ontology editing scenario. The implementation without blockchain cannot perform this scenario.

Operation	Time [s]	Gas used	Price [€]
A initializes the ontology	407 ± 13	705801 ± 0.0	2.47 ± 0.0
A executes DELETE query	16 ± 0.15	111777 ± 0.0	0.39 ± 0.0
B loads ontology from blockchain	311 ± 9.88	/	/
B executes INSERT query	18 ± 4.52	94677 ± 0.0	0.33 ± 0.0
A reads IDs from the blockchain	0.16 ± 0.02	/	/
A downloads data from IPFS	0.02 ± 0.0	/	/
A applies changes locally	1.06 ± 0.03	/	/
A executes a SELECT query	1.28 ± 0.26	/	/
Total	**736 ± 15**	**912255 ± 0.0**	**3.2 ± 0.0**

5.6. *Collaboration Between Users*

Our final test was designed to simulate collaborative editing of an ontology. We created two separate users A and B each with their own Ethereum wallet. The users then edited the ontology together. This scenario is constructed from previous scenarios. In the first step, user A initializes the ontology. The steps required for the initialization are shown in Section 5.3. User A then performs a delete query. The steps required for the delete operation are presented in Section 5.4. After that, user B reconstructs the ontology from the blockchain as is shown in Section 5.5. User B then executes an insert query adding an element to the ontology. The steps for the insert query are presented in more detail in Section 5.4. After that, user A updates their local ontology to the latest version by reading the identifier of the change performed by B from the blockchain, downloading the query from the IPFS network, and executing it on the local ontology. Finally, user A performs a select query to confirm that the change is visible. Times and prices of each operation are presented in Table 5. This scenario is also not possible if we are not using the blockchain. Because of that, we do not present the results for the basic Apache Jena implementation without our modifications.

Most of the time in this scenario is used for initialization of the ontology for both users. In real use, this is not a big issue, since the ontology only has to be initialized once. Another operation that requires a relatively long time is the execution of queries that change the ontology data. The long time is the result of limitations with Ethereum smart contracts. This can be improved by combining larger changes into one transaction. The operations that require only read operations are performed quickly.

In this test, each user had their own local Semantic database. It is also possible to share a single local Semantic database amongst multiple users. In this case, it is especially important that all changes to the content of the database are done through our transaction manager. If a user changes the database directly, the change does not get recorded on the blockchain so we get an inconsistency between the local ontology and the one stored on the blockchain. This inconsistency can be later resolved by restoring the ontology from the blockchain. In our implementation, we recommend the use of Apache Jena TDB as the

Table 6
Time and cost of each step in the consistency checking scenario. Some operations perform smart contract functions and require gas payment.

Operation	Time [s]	Gas used	Price [€]
Loading initial data to Jena	80 ± 2.92	/	/
Checking consistency	25 ± 0.83	/	/
Publishing the smart contract	17 ± 0.09	310149 ± 0.0	1.09 ± 0.0
Publishing initial data to IPFS	3.35 ± 0.13	/	/
Publishing identifiers	49 ± 5.26	301014 ± 0.0	1.06 ± 0.0
Perform SELECT query	0.18 ± 0.02	/	/
Perform DELETE query	0.02 ± 0.0	/	/
Checking consistency after update	151 ± 4.87	/	/
Upload query to IPFS	0.22 ± 0.01	/	/
Save identifier to Ethereum	16 ± 0.21	111777 ± 0.0	0.39 ± 0.0
Total	$\mathbf{492 \pm 11}$	$\mathbf{722940 \pm 0.0}$	$\mathbf{2.53 \pm 0.0}$

database for storing the ontology. However, the proposed transaction management system could also work with any other Semantic database.

Each change of the ontology gets executed as a transaction, where all changes get reverted in case of an error at any point. In the proposed implementation the changes are synchronized with the blockchain before each set of queries that the user requests. The synchronization is done by checking for new changes on the blockchain. After that, the new queries are loaded from the IPFS network and applied to the local database. Another option for the implementation would be to synchronize the local ontology with the one stored on the blockchain in real-time as the changes get made. In both cases, we get the latest version of the ontology before the new change gets applied. This is important since we could get an inconsistent state of the ontology if the local database would not contain the same information that is stored on the blockchain. Lastly, the transaction gets committed and new changes get recorded on the blockchain.

5.7. *Consistency Checking*

We performed an additional scenario to test the effects that consistency checking has on the performance of the system. In this test, we only loaded three out of four parts of our ontology, because consistency checking over the entire ontology was too memory intensive. For consistency checking, we used transitive rules. In this scenario, the user first loads the initial ontology and checks for consistency. The initial ontology is then uploaded to IPFS and the Ethereum network. After that, the user performs a SELECT and a DELETE query. These queries were selected to demonstrate the effect of consistency checking on a query that only reads the data and on a query that also changes the data. After the DELETE query, consistency is checked again. After that the update is also uploaded to the IPFS and to the Ethereum network. The time and payment required for each step of the process are presented in Table 6.

Checking the ontology consistency requires less time than the initialization of the ontology. On the other hand, we have observed that the memory required to perform a consistency check becomes an issue with large ontologies. The consistency check also has to

be performed after each query that changes the data. We believe that the benefits of using consistency checking outweigh the performance cost when editing smaller ontologies. When working with larger ontologies, for example, the entire DBpedia, the consistency checking is too demanding for practical use. This is not a limitation of our transaction manager, but rather of the Apache Jena framework.

6. Conclusion and Future Work

We propose a system for ontology management that uses the Ethereum and the IPFS networks to track changes and enable collaborative ontology building (Gašperlin, 2021). We tested our implementation on several scenarios and compared the results to an implementation without the distributed management part. Based on the results we conclude that the biggest bottleneck when using our system is the time required to change the smart contract on the Ethereum network. This means that each query that changes the data, takes between 15 and 30 seconds. This is a limitation of using Ethereum and can be circumvented by combining multiple changes into a single transaction since the time remains mostly the same regardless of the size of the transaction. By using the Ethereum smart contract we also get multiple advantages, that would be difficult to achieve otherwise. The system is completely decentralized and does not require any centralized servers or storage. We can track the history of every change that can not be changed by anyone. We can also track the author of each change in a tamper-resistant way.

One option for speeding up the operation might be to store the IPFS identifiers in another way. We could, for instance, store them in a file with a static address on the IPFS network. By doing so, the amount of time required to publish a transaction would decrease; however, we would lose the protection against changes to the recorded history. In addition to that, we would no longer have a tamper-resistant way of recording the author of each change.

In our current implementation, consistency is checked in the program before a change is sent to the smart contract. This means that an adversary could bypass the check and publish a change that breaks the ontology consistency. For future research, we propose a voting mechanism that would require enough trusted users to approve the consistency of a change in order for the change to get applied to the ontology. This would prevent anyone from applying inconsistent changes.

References

Apache Jena (2021). Reasoners and rule engines. https://jena.apache.org/documentation/inference/. [Accessed 13. 03. 2022].

Benet, J. (2014). IPFS – Content Addressed, Versioned, P2P File System. *CoRR*, *abs/1407.3561*. http://arxiv.org/abs/1407.3561.

Binance team (2022). Introducing BNB Chain: The Evolution of Binance Smart Chain. https://www.binance.com/en/blog/ecosystem/introducing-bnb-chain-the-evolution-of-binance-smart-chain-421499824684903434. [Accessed 11.03.2022].

Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J. (2008). Freebase: a collaboratively created graph database for structuring human knowledge. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 1247–1250.

Buterin, V. (2014). A next-generation smart contract and decentralized application platform. *White Paper*, *3*(37).

Cano-Benito, J., Cimmino, A., García-Castro, R. (2019). Towards blockchain and semantic web. In: *International Conference on Business Information Systems*. Springer, pp. 220–231.

Choi, S.-M., Park, J., Nguyen, Q., Cronje, A. (2018). Fantom: a scalable framework for asynchronous distributed systems. arXiv preprint arXiv:1810.10360.

Decker, S., Melnik, S., Van Harmelen, F., Fensel, D., Klein, M., Broekstra, J., Erdmann, M., Horrocks, I. (2000). The semantic web: the roles of XML and RDF. *IEEE Internet Computing*, 4(5), 63–73.

Donnelly, K. (2006). SNOMED-CT: the advanced terminology and coding system for eHealth. *Studies in Health Technology and Informatics*, 121, 279–290.

Edelman, G. (2021). The Father of Web3 Wants You to Trust Less. https://www.wired.com/story/web3-gavin-wood-interview/. [Accessed 30.5.2022].

Efanov, D., Roschin, P. (2018). The all-pervasiveness of the blockchain technology. *Procedia Computer Science*, 123, 116–121. https://doi.org/10.1016/j.procs.2018.01.019.

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

Gašperlin, D. (2021). Blockchain-based ontology database implementation. https://github.com/UL-FRI-Zitnik/blockchain-based-ontology-database. [Accessed 29.12.2021].

Haber, S., Stornetta, W.S. (1991). How to time-stamp a digital document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791.

Hector, U., Boris, C. (2020). BLONDiE: Blockchain Ontology with Dynamic Extensibility. *CoRR*, *abs/2008.09518*. https://arxiv.org/abs/2008.09518.

Horrocks, I., Patel-Schneider, P.F., Van Harmelen, F. (2003). From SHIQ and RDF to OWL: the making of a web ontology language. *Journal of Web Semantics*, 1(1), 7–26.

Jianjun, S., Ming, L., Jingang, M. (2020). Research and application of data sharing platform integrating Ethereum and IPFS Technology. In: *2020 19th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pp. 279–282. https://doi.org/10.1109/DCABES50732.2020.00079.

Khattak, A.M., Latif, K., Han, M., Lee, S., Lee, Y.-K., Kim, H.-I. (2009). Change tracer: tracking changes in web ontologies. In: *2009 21st IEEE International Conference on Tools with Artificial Intelligence*. IEEE, pp. 449–456. https://doi.org/10.1109/ICTAI.2009.42.

Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C. (2015). DBpedia – a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2), 167–195. https://doi.org/10.3233/SW-140134.

Liang, Y., Alani, H., Shadbolt, N. (2005). Ontology change management in protégé. In: *Proceedings of the 1st AKT Doctoral Symposium, June 2005*.

Markovic, M., Edwards, P., Jacobs, N. (2019). Recording provenance of food delivery using IoT, semantics and business blockchain networks. In: *2019 6th International Conference on Internet of Things: Systems, Management and Security*, *IOTSMS 2019*, pp. 116–118.

McBride, B. (2004). The Resource Description Framework (RDF) and its vocabulary description language RDFS. In: *Handbook on Ontologies*. Springer, Berlin Heidelberg, pp. 51–65. https://doi.org/10.1007/978-3-540-24750-0_3.

McGuinness, D.L., Van Harmelen, F. (2004). OWL web ontology language overview. *W3C Recommendation*, 10(10), 2004.

Naim, B.A., Klas, W. (2019). Knowledge graph-enhanced blockchains by integrating a graph-data service-layer. In: *2019 6th International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019*, pp. 420–427.

Nakamoto, S. (2009). Bitcoin: a peer-to-peer electronic cash system. *cryptography – The Cryptography and Cryptography Policy Mailing List*, https://metzdowd.com.

Nizamuddin, N., Salah, K., Azad, M., Arshad, J., Habib ur Rehman, M. (2019). Decentralized document version control using Ethereum blockchain and IPFS. *Computers & Electrical Engineering*, 76, 183–197. https://doi.org/10.1016/j.compeleceng.2019.03.014.

Noy, N.F., McGuinness, D.L. (2001). Ontology development 101: a guide to creating your first ontology. *Knowledge Systems Laboratory*, *32*.

Pérez, J., Arenas, M., Gutierrez, C. (2006). Semantics and complexity of SPARQL. In: *International Semantic Web Conference*. Springer, pp. 30–43.

Preston, I., LeDuc Díaz, C., Diesler, T. (2020). java-ipfs-http-client, GitHub. [Accessed 15.12.2021].

Shadbolt, N., Berners-Lee, T., Hall, W. (2006). The semantic web revisited. *IEEE Intelligent Systems*, 21(3), 96–101.

Shortliffe, E. (1976). Computer-based medical consultations: MYCIN. *Artificial Intelligence – AI*, 388. https://doi.org/10.1097/00004669-197610000-00011.

Siemer, S. (2019). *Exploring the Apache Jena Framework*. Georg-August-University Göttingen.

Speer, R., Havasi, C. (2012). Representing general relational knowledge in conceptNet 5. In: *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)* pp. 3679–3686.

Steichen, M., Fiz, B., Norvill, R., Shbair, W., State, R. (2018). Blockchain-based, decentralized access control for IPFS. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1499–1506. https://doi.org/10.1109/Cybermatics_2018.2018.00253.

Stevens, R., Goble, C.A., Bechhofer, S. (2000). Ontology-based knowledge representation for bioinformatics. *Briefings in Bioinformatics*, 1(4), 398–414.

Taye, M.M. (2010). Understanding semantic web and ontologies: theory and applications. *Journal of Computing*, 2.

Ugarte, H. (2017). *A More Pragmatic Web 3.0: Linked Blockchain Data*. Bonn, Germany.

T. Knez received his MSc degree in computer and information science in 2021. He currently holds the position of a young researcher at the Faculty of Computer and Information Science, University of Ljubljana. He is currently researching the use of knowledge bases in natural language processing.

D. Gašperlin received his MSc degree at the Faculty of Computer and Information Science, University of Ljubljana in 2021. In his work, he researched the use of blockchain technologies in the context of ontology construction. He is currently working as a software engineer at LeanIX.

S. Žitnik is an assistant professor at the Faculty of Computer and Information Science, University of Ljubljana. His main research interests are information retrieval and information extraction. Specifically, he is trying to enrich the extracted data from text using parallel and iterative combination of entity extraction, relationship extraction and coreference resolution techniques. Furthermore, his research also focuses on data merging, redundancy elimination and ontologies.

M. Bajec is a full professor at the Faculty of Computer and Information Science, University of Ljubljana. He is the Head of the Laboratory for Data Technologies and Iot Demo Centre. His research interests that primarily focus on IT and data Governance include: information retrieval, Web search and extraction, data integration, data management, data analysis, software development methods, IT/IS strategy planning. In his career he has led or coordinated over 30 applied and research projects. For his contribution in transferring knowledge to industry he received several awards and recognitions.