Leipzig University
Faculty of Mathematics and Computer Science
(Institute of Computer Science)

Trustworthy Provenance Recording using a blockchain-like database

Master's Thesis

Submitted by: Martin Stoffers

RegNo.: 3748896

Supervisor: Dr. Michael Martin, Mr. Andreas Schreiber

Topic suggestion: German Aerospace Center
Simulations and Software Technology
Linder Höhe
51147 Köln

Date: Thursday 30th March, 2017

Abstract

One of the research fields at the Institute for Simulation and Software Technology at the German Aerospace Center (DLR) , is to discover new concepts which provide reliability and trustworthiness in software applications [1, 2]. Since today's process operations in distributed systems are fairly complex, recording all actions is critical in order to ensure reliability and constitute trust in the systems.

One example application, developed by the department of Intelligent and Distributed Systems in collaboration with the German Space Operations Center (GSOC), is the Backend Catalog for Relational Debris Information (BACARDI) [3]. Its purpose is to collect and store information about more than ten million objects in orbit around earth, measured by various sensor networks. Before storing the data additional calculations like size, velocity, orbit, and collision detection with other objects, are performed on each object in a distributed manner. Collecting provenance about involved steps and actors is vital to prove the reliability of the events, especially to detect possible collisions. On the one hand it is about collecting provenance, on the other hand it is about protecting provenance from unintended or intended changes, since only unchanged data can be utilised for trustworthy predictions and safe inference in BACARDI or similar systems [3].

After Nakamoto proposed the blockchain technology in his paper, new types of tamper resistant and distributed data storage techniques have evolved from it [4]. These technologies can be applied to vital provenance data to protect it from possible alternation, even in distributed systems. Therefore, the main subject of this thesis is to elaborate and survey concepts for storing provenance data in blockchain-like databases. These concepts are analysed in terms of their advantages and disadvantages, tamper-resistance, and possible use cases, followed by a proof-of-concept implementation of all concepts in one particular blockchain-like database. Subsequently, performance measurements are conducted on each implementation followed by an analysis with respect to the concept design.

Acknowledgements

I would first like to thank my supervisors, Mr. Andreas Schreiber of the institute for Simulation and Software Technology at the German Aerospace Center (DLR) and Dr. Michael Martin of the faculty for Mathematics and Computer Science at Leipzig University. The door to their offices were always open whenever I ran into trouble or had a question about my research or writing.

I would also like to acknowledge Jeanine Watson, Anna Beer and Julia Fedorenko, who helped me as proofreaders of this document, and I am gratefully indebted to them for their very valuable comments and corrections on this thesis.

Finally, I must express my very profound gratitude to my parents, grandmother and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Martin Stoffers

Cologne, Thursday 30$^{\text{th}}$ March, 2017

Contents

Chapter 1

Introduction

1.1 Motivation and Problem Discussion

Today's fast growing, yet malleable data sets – which can be copied, altered or spread in various ways by new technologies – highlight the inevitable need to track the origin and lineage of such data, to provide trust. This is especially true, if data lineage about tasks and processes in distributed systems is collected and stored by many agents at the same time. Proof of integrity has a substantial part in creating trust in such data, since in this case only unchanged data is valid, and therefore, trustworthy for further and safe inference. A widely known method to check, if data in general has remained unchanged since it was stored or processed, is to calculate cryptographic hashes from the data, before saving or processing it. This hash is kept secret or announced publicly, depending on its usage. Later on, one can prove the integrity of the data by repeating the process and then comparing the result with the previously stored hash. If both hashes have the same value, the data can be considered as valid and, therefore, trustworthy.

However, "How to Break MD5 and Other Hash Functions" by Wang and Yu has shown that this process can produce hash collisions which might enable attackers to alter the data, but still ensure that the hash remains the same. In this case, one would consider the data as valid even if it had been changed. It is believed that current hash methods, like SHA256 or SHA512, are solving these problems. Yet, recent findings show that with steady rising computing power, these methods could be considered insecure as well [6, 7]. This is especially true if one wishes to guarantee the integrity of a growing data set over a very long period, which may exceed the save lifespan of the used hashing algorithms. An extended method to provide trust in data or hashes itself is to sign them with digital signatures. This technique, known as key signing, enables one to prove that only trusted third parties worked with the data and thus the data may be considered as trustworthy; still, neither of these methods is able to provide tamper-resistant data. With the invention of cryptographic Merkle-Trees by Merkle in his work "Secrey, Authentication, and Public Key Systems" in 1979, several storing concepts were proposed which led to more tamper-resistant data structures in general [8].

However, without having a byzantine fault tolerant system, a trustworthy data store in a distributed fashion is not achievable [9]. By combining peer-to-peer networking with Merkle trees, asymmetric cryptography, time-stamping and proof-of-work, Nakamoto proposed the blockchain as one solution [4]. His concept evidently creates trust and authenticity by providing a tamper-resistant distributed database for maintaining the ownership of digital money, including the complete lineage of all occurred transactions. Derivative concepts of distributed and tamper-resistant storage systems, with different aspects in mind, have evolved from this first blockchain.

Since provenance and blockchain are focused on the lineage of data the combination of both worlds may lead to more tamper-resistant provenance databases in distributed systems. The following chapters will propose an integrated approach to achieving tamper-resistant storage systems for provenance data with the use of blockchain-like databases.

1.2 Methodical Approach

In 2013, the World Wide Web Consortium (W3C) released the PROV standard for generating coherent and machine readable provenance data along with a data model [10, 11]. Based on its core definitions, this document will discuss and offer three concepts of how to secure provenance data by storing it into a tamper-resistant blockchain database. The analysis of these concepts will be done with respect to the following research questions:

What are the possible concepts for storing provenance in blockchain-like databases?

For each concept an explanation is given along with the mapping between the core PROV Data Model (PROV-DM) concepts and the blockchain database. Each mapping is presented with similar data sets to illustrate the differences between the concepts.

What are the influences of the concepts regarding tamper-resistance and security of the blockchain?

Each concept will be examined with respect to the usual blockchain concepts. This will highlight advantages and disadvantages in the concepts in terms of tamper-resistance and security.

What are the main differences between the concepts in terms of provided trustworthiness?

With answering the former questions a conclusion about the trustworthiness for each concept is given.

What are appropriate use cases for the concepts?

Each concept will be concluded with guidance on when to use which concept in a given environment, and use case.

1.3 Structure

Chapter 2 In the second chapter the term provenance, and the PROV standard is introduced. Further on, a distinction between the terms immutable and tamper-resistant is given. Thereafter, an explanation of the blockchain technology, including an analysis of its possibilities to store data, is given along with a discussion regarding tamper-resistance. Lastly, alternative blockchains and blockchain-like databases are discussed in the same context

Chapter 3 The third chapter discusses previous works, which have already proposed solutions to data storage in blockchains, or similar data structures. Hereafter, a distinction to this work will be provided to the reader.

Chapter 4 In chapter four the three concepts are proposed along with a discussion of the former mentioned research questions. The chapter is concluded with comparison of the three concepts.

Chapter 5 Chapter five first describes the requirements and definition, followed by the derived software architecture of concepts. Afterwards important aspects of implementation are highlighted for each concept.

Chapter 6 Chapter six first describes the test environment used to test the implemented concepts. Subsequently, the test procedure is explained. In the last section the measurements are analysed for each concept with a strong focus on performance. The chapter is concluded by a summary and a general comparison of the concepts.

Chapter 7 Finally, an overview about the achieved results is given and further research topics are mentioned.

Chapter 2

Background

2.1 Provenance

2.1.1 Definition

According to Oxford Dictionaries and Wikipedia, the term provenance originated from the French word provenir and can be translated as "to come from" [12, 13]. It was originally used in relation to works of art, describing the ownership and locations of an object [13, 12]. But nowadays the term is used in many research fields, including computer science. As stated by Moreau in "The Foundations for Provenance on the Web":

> "Provenance [...] is becoming an important concern for several research communities in computer science, since it offers the means to verify data products, to infer their quality, to analyse the processes that led to them, and to decide whether they can be trusted."[14]

Since this document focuses on the PROV standard to express provenance data, the W3C definition of the term provenance will be used:

> "Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness."[10]

2.1.2 Distinction to Metadata

The term metadata is used to describe objects by their properties, for example the size or author of a file [15]. Some of these properties are related to provenance. The file size is a property that gives direct information about the content itself, in contrast to the author who was involved in editing the file. In conclusion, provenance is as a subset of metadata, which only contains information describing the lineage of data [15].

2.2 PROV Standard and PROV Data Model

2.2.1 Definition

As introduced before, the PROV standard was released by the W3C Provenance Working Group, in April 2013 [10]. The PROV Data Model superseded the older OPM Provenance Model (OPM), which was originally initialised on the first International Provenance and Annotation Workshop (IPAW) and released to the public in December 2007 [16, 17, 18]. According to the W3C, the

Table 2.1: Overview PROV Attributes [11]

Attribute	Allowed In	Value
prov:label	Any construct	A value of type xsd:string
prov:location	Entity, Activity, Agent, Usage, Generation, Invalidation, Start and End	A value
prov:role	Usage, Generation, Invalidation, Association, Start and End	A value
prov:type	Any construct	A value
prov:value	Entity	A value

model's purpose is to translate domain or application specific provenance representations into a generic model. This in turn can be used to exchange, process, and reason over the data in heterogeneous systems, like the web [11].

In addition to the syntax of PROV-DM, the Provenance Working Group defined formal semantics which allow the mapping of the data model into different representations [19]. The most important are: PROV Ontology (PROV-O), PROV Extensible Markup Language (PROV-XML) and PROV Notation (PROV-N) [20]. PROV-O expresses the model with the Web Ontology Language in Version 2 (OWL2) and the Ressource Description Framework (RDF), while using the RL profile of OWL2 [21]. This representation is intended for reasoning over provenance data, while leaving five relations of the PROV-DM out of scope [21]. The PROV-XML representation is intended for exchange of provenance data across systems [22]. The same intention has recently submitted PROV JavaScript Object Notation (PROV-JSON) by Huynh and et al., which so far has the status of a member submission to the W3C [23]. PROV-N is a specialised notation which can be used to express provenance data in a more readable format [24]. It is mostly used in examples throughout the standard, as well as in this work. Another important element of the model are constraints, which are used to validate provenance data against the standard by a predefined set of rules [25]. Since PROV is used as foundation for mapping provenance data into a blockchain-like database, a detailed description based upon the *PROV-DM: The PROV Data Model* is provided in the next subsections [11].

2.2.2 Basic Elements

The PROV standard defines some basic elements which are used in any representation and data model along with the standard. These elements are firmly explained below. A more detailed description can be found in chapter 5.7 of the PROV Data Model Specification [11].

Namespaces Namespaces are identified by an Internationalized Resource Identifiers (IRI) as defined in RFC 3987 [26]. The PROV namespace defaults to the IRI `http://www.w3.org/ns/prov#` and maintains all of its own concepts, while avoiding naming collisions with other standards. For convenience in various representations the base IRI is mapped to a prefix, which for PROV is usually `prov`. For example, the absolute IRI `http://www.w3.org/ns/prov#agent` is then mapped to `prov:agent`.

Qualified Name According to PROV-DM, qualified names consists of a namespace, which could be shortened by a prefix, and a local name. If no namespace or prefix is denoted it refers to the default namespace. Therefore, a qualified name must be a valid IRI, but not necessarily in the PROV namespace.

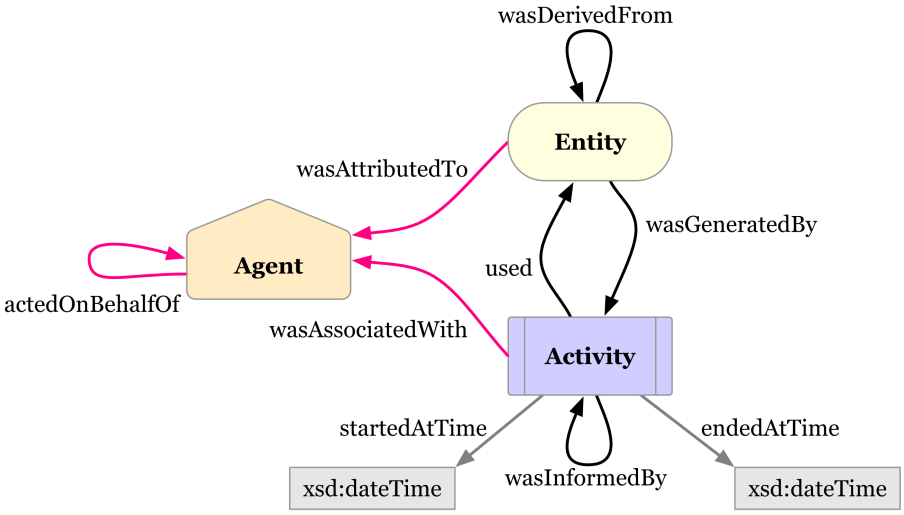


Figure 2.1: PROV-DM Core Classes [21]

Identifier Identifiers are defined as valid qualified names and considered unique. Hence, if one identifier appears more than once in a document it must be the same thing or object.

Attributes and Values Attributes are used to attach further information to objects. These attributes are predefined in the PROV namespace (Table 2.1). Other attributes, that do not follow this naming convention are not interpreted. Values assigned to attributes are constants of type, string, number, time, qualified name, IRIs or encoded binary data [11]. However, the standard itself recommended preferring compatible RDF types or qualified names [11, 27].

2.2.3 Types

As briefly mentioned in the definition of the W3C, the use of PROV-DM is to express the process of generating entities with activities by involved agents. The types *entity*, *activity*, and *agent* together with the relations between them, form the core concepts of the PROV-DM (Figure 2.1). Since every component is modelled around these three types, a description along with each respective formal PROV-N notation is given below.

Entity An entity is the prime type and concept; it is considered the object or more general thing one wants to describe provenance about. In terms of the model an entity can be a physical, digital or conceptual thing, as long as one can collect provenance about it [11]. As depicted in Figure 2.1, an entity is usually denoted as a yellow coloured ellipse. The entity is expressed in PROV-N as `entity(id, [attr1=val1, ...])`, where `id` is the unique and mandatory identifier followed by an optional set of attribute-value pairs. [11] The example states an instance of an entity with the identifier `ul:thesis-stoffers`, prov:type of `ul:mastersthesis`.

```
entity(ul:thesis-stoffers-16032017,
       [prov:type="ul:mastersthesis"])
```

Activity The type activity represents actions performed upon entities, while other entities may be created in this process. Possible actions, among others are: generating, transforming, or modifying one or more entities [11]. As shown in Figure 2.1, an activity is usually denoted as a blue rectangle.

An activity is expressed in PROV-N as `activity(id, st, et, [attr1=val1, ...])`, where `id` is the unique identifier of that activity followed by optional attributes. `st` and `et` describe the start and end time of an activity, while the set of attributes has the same purpose as in an entity. The following example states an instance of an activity with the identifier `ul:worked-on-thesis` and prov:type `ul:edit`. It was started and completed on the 3th of October 2016.

```
activity(ul:worked-on-thesis, 2016-10-03T09:00:00,
         2016-10-03T09:30:00, [prov:type="ul:edit"])
```

Agent An agent can be considered to be a person or machine who is responsible for an activity. Therefore, an agent is capable of influencing entities or acting on behalf of other agents. As depicted in Figure 2.1, an agent is usually denoted as a orange house shaped polygon. An agent is expressed in PROV-N as `agent(id, [attr1=val1, ...])`, where `id` is the unique identifier followed by an optional set of attribute-value pairs. The following example states an instance of an agent with the identifier `martin-stoffers`. The lists of attributes consist of a name `Martin Stoffers`, a `prov:type` of value `prov:Person` and a registration number of value `3748896`. While `prov:type` is located in the PROV namespace, the attributes `ul:regno` and `ul:name` are application specific.

```
agent(ul:martin-stoffers, [ul:regno="3748896",
      ul:name="Martin Stoffers", [prov:type="prov:Person"])
```

2.2.4 Components

The PROV standard is categorized into six components, which cover different aspects of provenance. A short overview about all components, types and relations is given in Table A.1 attached to Appendix A. The first three components wrap the core model, while the last three components are part of the extended model. The first component deals with entities, activities and relations to represent generation and usage of entities. Relations for defining start and end of an activity, as well as invalidation of entities, complete this component. The next component explains how to model the derivation of entities from others. The third component focuses on agents and their responsibility and influence on entities, activity and other agents. The fourth component introduces the concept of bundles, which enables provenance about provenance. The fifth component five describes how to model alternates of entities. The last component introduces collections to provide a logical structure to model provenance for groups of entities.[11]

Component 1

The first component of the PROV-DM consists of six concepts, which are expressed as relations between entities and activities. A Unified Modeling Language (UML) representation of this concept is depicted in Figure A.1 attached to Appendix A.

Generation Generation is the concept of producing entities with activities. As defined by the standard, this entity does not exist before its generation and thus cannot be used by another activity beforehand. Therefore, the generation is considered a relation between both types and is expressed in PROV-N as `wasGeneratedBy(id; e, a, t, attrs)`. The only required attribute is the identifier for the generated entity e itself. While each entity must have been created by an activity a, it is likely that one does not have information about the other attributes. To allow recording of such processes the standard declares all other attributes as optional, although at least one must be present. The following example expresses the generation of

`ul:thesis-stoffers` by activity `ul:worked-on-thesis`, while the exact time t and further attributes are undefined.

```
wasGeneratedBy(ul:thesis-stoffers-20170316,
               ul:worked-on-thesis, -)
```

Usage While generating new entities, an activity might use others in this process. For that reason the concept of usage is defined in PROV. Similar to generation, it is a relation between these two types. Its representation is written as `used(id; e, a, t, attrs)`. The specification of all attributes follows along the rules of generation. An example might be the usage of entity `tr:WD-prov-dm-20111215` in activity `ul:worked-on-thesis` at a given time t.

```
used(tr:WD-prov-dm-20111215, ul:worked-on-thesis,
     2017-03-16T09:10:00)
```

Communication The concept, communication, expresses the dependency of one activity onto another, established by the creation of an entity.

The PROV-N is written as `wasInformedBy(id; a2, a1, t, attrs)` and implies the exchange of an entity generated by activity *a1* and used by *a2*. While both activities must be declared in the attributes, id and time t are optional. The following example states that the activity `ex:print-thesis` is informed by `ul:worked-on-thesis`, as an entity is generated.

```
wasInformedBy(ex:print-thesis,
              ul:worked-on-thesis)
```

Start As previously outlined, an activity can depend on entities, generated by different activities. This, however does not imply the existence of that particular activity. Therefore, the start and generation of an activity can be expressed with the relation `wasStartedBy(id; a2, e, a1, t, attrs)`. The only required attribute is the identifier of activity *a2*, which was being triggered by entity e generated by activity *a1* at time t. Following the previous examples, a valid relation could be the start of an activity `ex:print-thesis`, triggered trough entity `ul:thesis-stoffers-20170316`. The activity involved in generating the entity can be set as third attribute, which in this case is the activity `ul:worked-on-thesis`.

```
wasStartedBy(ex:print-thesis, ul:thesis-stoffers-20170316,
             ul:worked-on-thesis, 2017-03-17T10:00:00)
```

End Similar to the start of an activity the standard defines a relation for its end of existence. The end of an activity is triggered by an entity, which was in turn is generated by an activity. The PROV-N notation is written as `wasEndedBy(id; a2, e, a1, t, attrs)`, while all attributes are defined as in the wasStartedBy relation. Therefore, the only difference in the example, beside the relation name, is the time when the activity `ex:print-thesis` was ended.

```
wasEndedBy(ex:print-thesis, ul:thesis-stoffers-20170316,
           ul:worked-on-thesis, 2017-03-17T10:10:00)
```

Invalidation The concept of invalidation describes the process of destruction, or expiring of an entity, by an activity. After invalidating an entity it is no longer available for further use, but this does not necessary state a complete deletion. It is defined as `wasInvalidatedBy(id; e, a, t, attrs)`, while only the entity *e* itself must be declared in the attributes. As shown in the example, the entity `ul:thesis-stoffers` is invalided by the activity `ul:print-thesis`, since the generated entity is the new valid version of the thesis.

```
wasInvalidatedBy(ul:thesis-stoffers-20170316, ex:print-thesis, -)
```

Component 2

The second component describes four concepts related to derivations of entities from other entities. As one can see in Figure A.2 attached to Appendix A, all concepts are defined as relations between entities.

Derivation, Revision, Quotation and Primary Source The process of derivation can be determined as a result of generating a new entity as an update, revision or quotation of another entity. This includes the usage of other entity as a main source in the process. According to the standard and explained above, this implies an underlying activity. Any kind of derivation must be declared with the statement `wasDerivedFrom(id; e2, e1, a, g2, u1, attrs)`; where the generated entity *e2* and the used entity *e1* must be declared, while all other attributes are optional. These attributes are: the activity *a* generating the entity *e1* and, if known, relations for generation *g2* and usage *u1*. Specific kinds of derivations are expressed trough a `prov:type` entry in the optional attributes. As given in the example, a revision of `ul:thesis-stoffers-20170310` can be sub-typed from a derivation using the prov:type `prov:Revision`.

```
wasDerivedFrom(ul:thesis-stoffers-20170316,
               ul:thesis-stoffers-20170310, -, -, -,
               [prov:type="prov:Revision"])
```

Component 3

The third component introduces the type agent and relations to model its responsibility for entities, activities or other agents. Further on, more general relations to express influence are defined. The four underlying concepts are depicted in Figure A.3 and Figure A.4.

Attribution An agent is attributed to an entity if he was involved in the process of generating it by using an activity. According to the standard, attribution is expressed as `wasAttributedTo (id; e, ag, attrs)`. While entity *e* and agent *ag* are mandatory, all other attributes are optional. The following example states that the entity `ul:thesis-stoffers-20170316` was attributed to the agent `ul:martin-stoffers`.

```
wasAttributedTo(ul:thesis-stoffers-20170316, ul:martin-stoffers)
```

Association Similar to the attribution, the concept association expresses the responsibility of the agent, but in terms of activities. The relation is defined as `wasAssociatedWith(id; a, ag, pl, attrs)` with activity *a* as mandatory attribute, while all others are optional. The plan *pl* is defined as entity with the prov:type `prov:Plan`, which the agent *ag* relies on to achieve some goals. The following example states that agent `ul:martin-stoffers` is associated with activity `ul:worked-on-thesis`, while executed it as part of the plan `ul:milestone-1`.

```
wasAssociatedWith(ul:worked-on-thesis, ul:martin-stoffers,
                  ul:milestone-1)
```

Delegation Delegation models the concepts of responsibility and authority of agents on others. The relation, written as `actedOnBehalfOf(id; ag2, ag1, a, attrs)`, states that agent *ag2* acted on behalf of agent *ag1*. The activity *a* represents the process, which agent *ag2* executed. While *ag1* and *ag2* are mandatory attributes, *a* and *attrs* are optional. A simple example is listed below.

```
actedOnBehalfOf(ex:print-shop, ul:martin-stoffers,
                ex:print-thesis)
```

Influence Some entities, activities, or agents may have influence on others which cannot be modelled with the previous concepts. Therefore, the standard introduced the wasInfluencedBy relation, which allows the linking of two types. The relation is written as `wasInfluencedBy(id; o2, o1, attrs)`. The attributes Object *o1* and Object *o2* could be any of entity, activity, or agent. They are mandatory, while id and attrs are optional. An example might be the indirect influence of the agent `ul:michael-martin` onto the entity `ul:master-thesis-stoffers` in his role as supervisor.

```
wasInfluencedBy(ul:master-thesis-stoffers, ul:michael-martin,
                [prov:role="ul:Supervisior"])
```

Component 4

Component four relates to the concept of bundles, which are useful if one wants to express provenance about provenance. The UML representation is pictured in Figure A.5.

Bundle constructor A bundle is defined like a normal entity, but extended with the attribute `prov:type` of value `prov:Bundle` in its attribute list. These entities are then used to link or merge previously collected provenance to or with other provenance data. For example, all provenance beforehand can be represented by the bundle `ul:master-thesis-stoffers`. Later on this bundle, representing the past work flow on the thesis, was attributed to the agent `ul:university-leipzig`.

```
bundle ul:master-thesis-stoffers

    entity(ul:master-thesis-stoffers, [prov:type="prov:Bundle"])
    entity(ul:thesis-stoffers-20170316,
           [prov:type="ul:mastersthesis"])

    wasAttributedTo(ul:master-thesis-stoffers,
                    ul:university-leipzig)
endBundle
```

Component 5

The fifth component introduces concepts for alternation and specialisations of entities, which are intended to highlight different aspects of the same object in provenance. The concepts are depicted in Figure A.6.

Specialization A specialised entity shares all aspects with the more generalised entity. The concept is close to a derived class in Object-oriented programming (OOP), which inherits all attributes from its parent class. In distinction to the concept derivation, no activity is directly involved, hence it's the same object. Specialisation is written as `specializationOf(infra, supra)` in PROV-N. An example, is a specific version `ul:thesis-stoffers-20170316` of a more generic entity `ul:thesis-stoffers-finished`.

```
specializationOf(ul:thesis-stoffers-20170316,
                 ul:thesis-stoffers-finished)
```

Alternate The concept, alternate, in contrast to specialised, can be used to link two entities which share some general aspects but are not considered the same object. An alternate entity is not influenced by the other entity and is therefore written as `alternateOf(e1, e2)` without further attributes. For example, entities of the same document generated in different processes can be stated as alternates, since both have their own provenance.

```
alternateOf(ul:thesis-stoffers-20170316,
                        ul:thesis-stoffers-20170310)
```

Component 6

Component six, shown in Figure A.7, introduces the concept of collections and how to model membership for it. They are intended to express general provenance for large sets of entities. This includes evolving of collections itself.

Collection and Membership A Collection is defined as an entity with the additional attribute `prov:type` of value `prov:Collection`, hence provenance can be recorded about it. A new member is put into the collection with the `hadMember(c, e)` relation. An example is a collection with all versions of the master thesis, which were generated throughout the writing process.

```
entity(thesis-stoffers-versions,
       [prov:type="prov:Collection"])

hadMember(ul:thesis-stoffers-versions,
          ul:thesis-stoffers-20170316)
hadMember(ul:thesis-stoffers-versions,
          ul:thesis-stoffers-20170310)
```

2.3 Immutability and Tamper-Resistance

Since blockchain databases are a new field of research, many claims are taken about their capabilities. One of these claims is the immutability of data in blockchains. As mentioned in the introduction, protecting provenance data from being altered is a big concern. To avoid

inaccuracy, this section will give a distinction between immutability and tamper-resistance, and how these terms are further used in this document.

According to Hasan and et al. in "*The Techniques and Challenges of Immutable Storage with Applications in Multimedia*" immutability is a characteristic or property given to an object. It forbids, by definition, all changes to that object over time. In terms of data, this would imply that no changes can be made since the data is permanent [28, 29]. As presented in "Tamper-Resistant Storage Techniques for Multimedia Systems" by Haubert et al., software based solutions are not capable of providing complete protection against tampering if the storage medium is not immutable. Thus, thinking about blockchains as distributed databases with software based protection mechanisms and algorithms, the characteristic of immutability is not attributable.

By design the blockchain can be considered as strong tamper-resistant, since it's very difficult to modify the data it contains. In contrast to immutability, the property tamper-resistance can range in levels of difficulty and, therefore, in the trustworthiness it provides.

As in other systems, these levels are bound to other parameters like performance or usability. Therefore, the term tamper-resistant is used further in this document as an indicator for the trustworthiness of a blockchain, with respect to other aspects.

2.4 Blockchains

2.4.1 History of Blockchains and Bitcoin

In 2008, Nakamoto proposed the blockchain in his paper "*Bitcoin: A Peer-to-Peer Electronic Cash System*" [4]. His goal was to create a public distributed ledger to account transactions of digital money without a centralised authority. To achieve this goal, Nakamoto used several techniques. As mentioned in the introduction, beside peer-to-peer networking, these are: asymmetric cryptography and digital signatures, time-stamping, proof-of-work, and Merkle-Trees [4, 31]. The combination of these techniques significantly reduce the possibility of double-spending and also provide a byzantine fault tolerant system [4, 32].

The first implementation of the Bitcoin blockchain and its inherent digital currency Bitcoin (BTC), was made public as open source software in 2009. In its current version 0.13.2, the blockchain has evolved into a peer-to-peer network of approximately ten million nodes which exchange about 230.000 transactions per day [33, 34]. The backbone of the network consists of about 5300 full nodes [35], which maintain a database with approximately 100 GByte in size. As of January 2017, Bitcoin is by far the largest public blockchain [34]. Since most of its successors are strongly influenced by Bitcoin, the following sections will highlight the common function principles and the basic concepts which ensure tamper-resistance and reliability in blockchains. A shorter overview, including storage capabilities, can be found attached in the appendix(Table B.1).

2.4.2 Function Principles

Peer-to-Peer Network

As mentioned above, the Bitcoin network is built by many nodes, which are directly connected to each other. In general, nodes are used to broadcast received messages with information about transactions and blocks to all its peers. The decision whether a message is relayed or not depends on the consensus rules of the network. One important rule is that honest nodes only broadcast messages containing a valid transaction or block [36] [37, Ch.6]. To check the validity, full nodes must maintain a database of all blocks and transactions they ever considered valid [37, Ch.6]. All other nodes must request missing information from these full nodes. Therefore, full

nodes are able to exchange information about requested transactions and blocks with its peers directly. These requests are often made by new full nodes entering the network or Simplified Payment Verification (SPV) clients, which do not maintain a full copy of the blockchain [37, Ch.6] [32]. The latter mentioned node type is mostly used in devices with limited resources, such as smartphones. Another vital node type for the network is the Bitcoin core client, which includes a mining component. Beside having a copy of the blockchain and broadcasting transactions and blocks, the node can add new entries to the blockchain [37, Ch.6].

Bitcoin Addresses

In order to send or receive Bitcoins to or from others, Bitcoin uses unique addresses. As shown in Figure B.2, an address is based upon a 256 bit cryptographic key pair generated by using the Elliptic Curve Digital Signature Algorithm (ECDSA). For anonymity reasons and size optimisation, the public key is transformed into a public key hash by calculating the hash with `RIPEMD160(SHA256(pubkey))` [38],[37, Ch.4]. The hash is completed by adding a single byte to the beginning of the address and a four byte checksum at the end. Finally, the result is Base58Check encoded for an easier exchange and usability [39].

Wallets

To create new transactions a node must have access to a wallet with one or more Bitcoin addresses [37, Ch.4]. Consequently, a wallet can create addresses and store the corresponding private keys [40]. To keep up with the current state of the blockchain, the wallet software monitors the Bitcoin network for new blocks and transactions [37, Ch.4]. If a transaction including a maintained address is found, the Bitcoin balance is updated and the transaction data is stored for later use [37, Ch.4]. In addition, the wallet tracks how many blocks do confirm each received transaction.

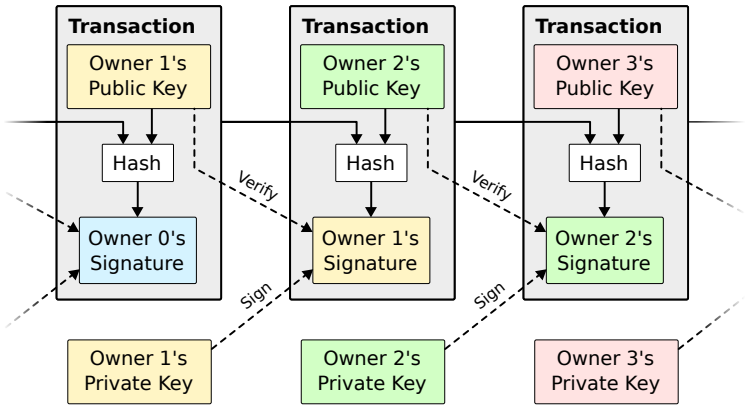


Figure 2.2: Transactions Concept [4]

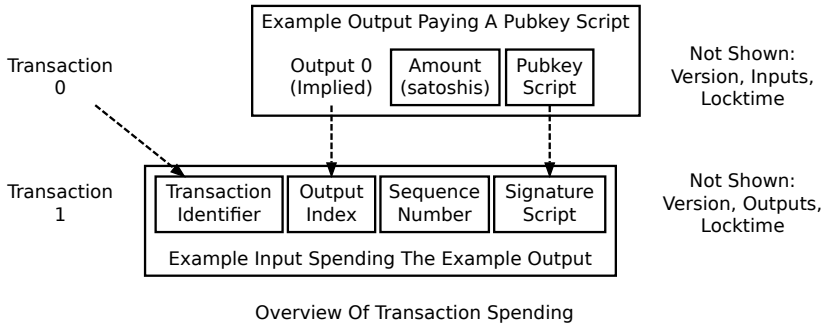
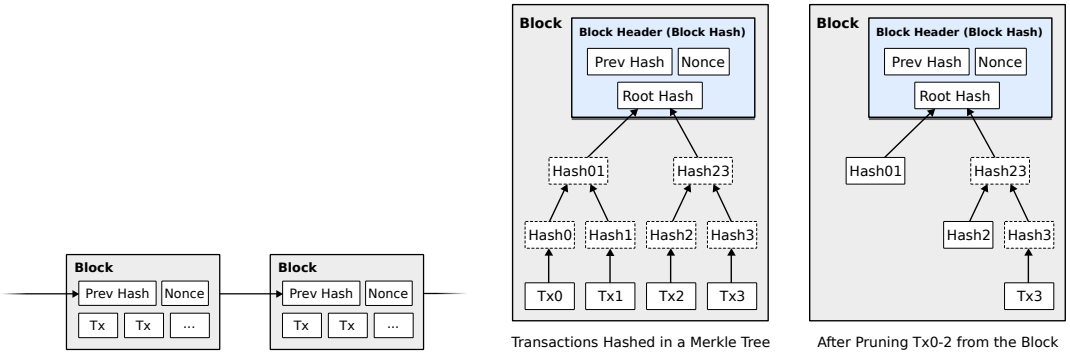


Figure 2.3: Bitcoin transaction [32]

Transactions

As depicted by Nakamoto in his paper, a coin and its lineage of ownership is recorded in a chain of digital signatures, also known as transactions (Figure 2.2) [4]. A simplified transaction must include: the amount of coins to transfer, the hash of the previous transaction, and the public key of the new owner [4]. By signing the transaction, the previous owner proves that he is the valid owner of the coin in the previous transactions and is willing to transfer it [4]. By linking the previous transaction hash, every client in the network can validate the ownership history by following the chain of transactions. Since the original transaction concept does not allow for splitting or combining Bitcoin values, or even get a change returned, the Bitcoin blockchain introduced a more complex transaction model with multiple transaction inputs and outputs (Figure B.1) [32]. To combine smaller amounts of Bitcoins multiple inputs are used, while multiple outputs are used for spending fractions of Bitcoins to multiple recipients or returning a change [32]. As long as nobody claims an output from a previous transaction by referencing it in a transaction input it is called Unspended Transaction Output (UTXO) [41].

When broadcasting a transaction into the network the ownership transfer is announced publicly. Other nodes in the network, which are receiving a message containing a transaction, must perform multiple checks on it before relaying them to their peers. These rules are explicitly listed in the *Protocol rules* in the developer section of the BitcoinWiki [42]. Beside checking if a transaction is already in the local transaction pool or included in a block, a node performs checks on syntactical correctness and applies rules regarding size and type of transaction. A



(a) Linking of blocks (b) Full block structure

Figure 2.4: Blocks in Bitcoin blockchain [4]

node also searches for each referenced UTXO in its transaction pool. If a UTXO is already spent in another transaction or is missing from the pool, the whole transaction is rejected. A node must ensure that a transaction does not spend more Bitcoins as assigned to the inputs. Consequently, the sum of all Bitcoins in the inputs must be greater or equal to the sum of all Bitcoins in the outputs [42]. If not all the Bitcoins are assigned to the outputs, the difference is collected as a fee by the miner. This will be explained later in more detail.

In contrast to the proposed transaction model, each UTXO also includes a PubKey Script instead of using plain public keys (Figure 2.3) [32]. By combining it with the signature script from the corresponding transaction input, a complete Script is formed [32, 41]. It is written in a Forth-like stack-based scripting system, which does not support loops due to stability and security reasons, hence it is Non-Turing complete [43]. A typical use case of Script is the verification of a public key hash from the Bitcoin address against a signature and public key in a pay-to-pub-key-hash transaction. The example script first puts the two values from scriptSig onto the stack (Listing 2.1). The opcode OP_DUP duplicates the top stack value, which is in this case the public key. Afterwards the top value is hashed with the function defined by OP_HASH160. After putting the public key hash onto the stack, both hashes are compared due to the OP_EQUALVERFIY opcode. Finally, OP_CHECKSIG verifies the signature against the public key and returns true if they match. Assuming multiple UTXOs as inputs in one transaction the verification will only succeed if each Script returns true [42]. Consequently, Script ensures the proof of ownership for any transfer and adds flexibility for other use. One usage of interest, is the capability of storing data, which will be explained in the next section. A transaction which is considered valid by a node it is added to the transaction pool and broadcast to its peers.

```
scriptSig:    <sig> <pubKey>
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Listing 2.1: Example script [43]

Blocks

While chained transactions are provide a strong proof of ownership, they do not solve the entire double-spending problem. In terms of digital assets or currencies, double-spending is the attempt to spend the same asset or coin more than once [44]. This problem can occur in a blockchain, if the same UTXO is used in more than one transaction as input. To avoid such attempts, Nakamoto introduced blocks to mark UTXOs as spent by time-stamping them [4]. Similar to

transactions, blocks are linked to the predecessor by its hash in the block header (Figure 2.4a). This chain of blocks can be considered the main database of any blockchain. A block header also includes a root hash (Figure 2.4b). Its value is calculated by hashing any transaction into a cryptographic Merkle-Tree, which should be time-stamped by the block. This well known technique was invented by Merkle in his work "Secrey, Authentication, and Public Key Systems" [8]. As stated in the white paper, the Merkle-Tree serves for two main purposes in Bitcoin [4]. First, it is used to reduce the size of the block header by adding only one hash value rather than the hash values of all related transactions. Second, it provides a tamper-resistant and easy to evaluate data structure for the database. The tamper-resistance is achieved due to the fact that the root hash of the tree will change if one of the transactions on the leafs is altered. The next value included is the nonce, which has an important role for the solution of the double-spending problem within the Proof-of-Work (PoW) mechanism in the mining process of new blocks [45, 32, 46]. In addition to the block concept of the original paper, the Bitcoin blockchain added three more values to the block header [46]. These are: a 32 bit integer for the version, a 32 bit unix timestamp, and the 32 bit nBits value which is also used in the PoW mechanism [46]. Its value is changed due to the mining power in the network, to ensure a constant block rate of approximately one block in ten minutes [46].

Mining blocks and Generating coins

As firmly introduced at the beginning of this section, a new block is generated by a full node with mining capabilities and a full copy of the blockchain. To successfully mine a new block, a node tries to create a new block on top of its longest chain of blocks [4, 32]. Subsequently, a coinbase transaction is generated by the miner and the hash is added to the Merkle-Tree [32]. This special transaction type is used to collect the mining fees from all included transactions and claim a reward for the new block. Afterwards, a node selects transactions by its own rules from its internal transaction pool, which are yet not included in any seen block. The hashes of those transactions are added into the Merkle-Tree, until the whole block size reaches about one megabyte in size. By hashing all header values a node tries to find a valid block header hash, which is less than or equal to the hash calculated from the nBits value [46]. Therefore, the node utilizes the four byte nonce to find a valid solution by changing its value, before including it into the hash algorithm [45]. Due to the 2^{32} bits, a node may have to try 4294967296 values in the worst case. This mechanism for mining a new block is the widely known Proof-of-Work [47]. Its main characteristic is that finding a solution to the problem is relatively hard, but validating the result by others is easy. If a mining node has found a solution, the block header is broadcast to its peers. Similar to transactions, other nodes must validate new block headers before relaying them to their peers. Accordingly, to the rules a node verifies that the block is not a duplicate of any other block in current chain by searching the block header hash in its database [42]. The new block header hash is checked against the current nBit value to ensure the block meets the desired difficulty. Afterwards, checks on syntactical correctness and time conditions are performed. If the first transaction is coinbase and the following list of transactions is not empty, a node must verify each transaction. These checks are similar to the checks performed on transactions itself. In case a transaction includes an already spent transaction output, the new block is invalid and dropped by the node. In addition, a node recalculates the Merkle root hash for comparison with the one in the new block header. To increase the processing speed of the Merkle-Tree, a node can utilize the Merkle-Proof without having all transactions in its transaction pool or requesting it[8]. One important rule to avoid the double-spending problem is to prove that the PoW was performed correctly by the node which mined the block. By hashing the calculated Merkle root hash with the given nonce and the previous block header hash, the node can verify the PoW by comparing the result with new block header hash [47, 42, 37].

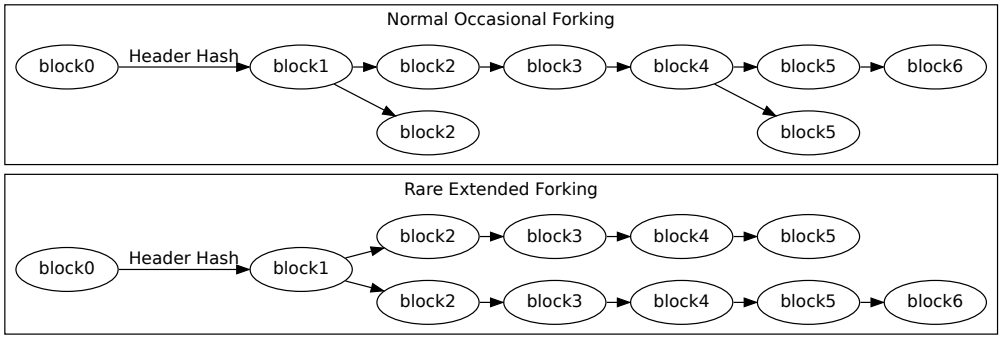


Figure 2.5: Forks in Blockchains [32]

Forking

In general forking refers to the point that a blockchain can be split in two branches at the last block, while the chain is built upon it [32]. The two forking types are *Normal occasional Forking* and *Rare extended Forking* (Figure 2.5). As shown in the example, multiple nodes mined a new block on top of *block 1* in the blockchain in both cases. Consequently, more than one valid block header for *block 2* is broadcast into the network. Due to signal delays the majority of nodes would not receive the same block headers at the same time. Therefore, each node may receive a different valid *block 2* first. According to the rule that only the longest chain is accepted by the network, nodes immediately start to mine on this block [42]. Valid blocks for *block 2* which are received later are added as forks. Until one node successfully finds a valid *block 3* on top of its *block 2*, it will relay the headers into the network. On receipt, all nodes still mining on top of its *block 2* will immediately switch to the fork with *block 3* to maximize the probability of gaining the next reward for block 4. This simple concept ensures linear history of blocks which all nodes can trust in.

Instead of normal occasional forking, rare extented forking can have two reasons; where one is the majority attack, explained in the next section [32]. The second cause of rare forks, also known as hard forks, are changes to the protocol of the blockchain which might be introduced as part of the normal development process. Since not all nodes are updated at the same time, they consider different blocks as valid. This leads to forks in the chain, were each version of the network is mining another branch of the blockchain. During the update process more and more nodes will change to mine on the newer fork, leaving an old stale fork. Since hard forks can include rules, forcing nodes to rebuild the blockchain from a specific block on, they can be considered problematic in terms of reliability and tamper-resistance. In August 2008, this occured in Bitcoin due to a critical exploit, which was actively used by attackers [48]. In consequence to the roll back, all transactions after the attack were deleted from the active blockchain.

2.4.3 Majority Attacks

Considering a network of honest mining nodes which are competing with each other to find the next block for claiming the reward, a possible attacker who tries to tamper with the blockchain must mine new blocks faster in order to attack the network [47, 49]. Such an attack becomes likely in Bitcoin, if an attacker owns more than 50 percent of all mining nodes in the network [4]. In this case it is possible to double-spend Bitcoins by first building a transaction with the correct payment address. As soon as the transaction is included in a block and recognized by the victim, the attacker broadcast another transaction with a different payment address but the

same UTXO as input into the network [49], [37, Ch. 8]. At the same time he announces a block on the same previous block, which includes the second transaction. This leads to a branch in the blockchain were the next block on top decides which branch will be the longest chain. Due to its mining capabilities, the attacker will successfully mine a block on top of its own block. Consequently, the first transaction is invalid by the rules of network and the victim is never paid. To prevent such an attack in the first place, a common rule is to consider a transaction as successfully stored in the blockchain if a certain amount of blocks are mined on top of the block, with the time-stamped transaction. Currently, a transfer is considered completed after six blocks on top, since the probability of an attack to happen at this point is under 0.1 Percent [50, 49]. Secondly, due to the current size of the Bitcoin network and the costs for setting up nodes with the capability of mining each new block does make an attack even more unlikely. In addition, mining blocks according to the rules with such capabilities would be more profitable due to the rewards [37, Ch. 8]. Therefore, a blockchain with a huge decentralized network of miners can be considered more tamper-resistant and trustworthy than networks with less mining capabilities or networks with centralized mining capabilities.

2.4.4 Altcoins

Driven by the success of Bitcoin, other digital currencies evolved from the first blockchain. The majority of these Altcoins are direct forks of Bitcoin's original implementation. With almost all functionality in common, changes usually aim for: faster transaction processing, different PoW mechanisms, new transaction types and coins supply strategies [37, Ch. 9]. Due to the function principles of blockchains these parameters are strongly coupled together. Consequently, changing them may have an influence in tamper-resistance and therefore in trustworthiness.

For example, Litecoin and Dodgecoin changed the PoW to prevent a centralisation of the mining power in huge pools like in Bitcoin [51, 52]. Therefore, both blockchains have started to use a simplified version of Scrypt as PoW mechanism [53, 54]. Instead of being computational intensive, Scrypt is designed to demand significant amounts of memory for the calculation. This idea should prevent miners from using Application Specific Integrated Circuit (ASIC)s for the computation of new blocks and enable slower computers to equally participate in the process. The creators of both Altcoins assumed that this will lead to a significantly better distribution of coins held by the miners. However, due to new specialised ASICs with larger memory and GPU computation, this concept showed similar problems to PoW.

Another approach is the Proof-of-Stake (PoS) mechanism [55]. One usage is present in the Peercoin blockchain, which aims to develop a more secure solution to majority attacks [56, 57]. Therefore, Peer-Coin introduced two types of blocks with different trust levels which are mined by PoS or PoW. Blocks mined with PoS do have a significantly higher trust level than PoW-Blocks. Consequently, nodes mined with PoS are preferred. By mining such a block a node places a special coinstake transaction, which includes unspent coins from the miner as input. To be a valid transaction the coins placed must have been unspent for a longer time period. The reward for the miner is an additional percentage of coins which is calculated form the coins spent. This concept is close to interest rates in fiat money. As a result miners without mining pools, but some coins, can successfully generate new blocks. By this means, the centralization of mining capabilities is reduced in comparison to Bitcoin.

In addition, blockchains like Dodgecoin or Litecoin, raised the block rate up to ten times faster than in Bitcoin [51, 52]. By raising the rate these blockchains provide a faster processing of transactions into blocks resulting in more efficient storage of data, but double-spending becomes more likely to happen. This is especially true, if such blockchains are using a similar concept for the Proof-of-Work mechanism as in large blockchains like Bitcoin. In combination with smaller or centralized networks of mining nodes, these chains should be considered less tamper-resistant.

2.4.5 Private Blockchains

A young field of research in blockchains are private or permissioned blockchains. Distinct from blockchains like Bitcoin, these chains are not open to the public and mostly intended for use in a controlled environment. They are usually deployed in private networks or networks owned by one or more contractors. The focus is to share reliable information about ownership and related transfers by using a blockchain data structure. By having a trustworthy group of participating nodes these chains usually do not use a PoW mechanism for validating the blocks. Omitting the PoW leads to a significantly faster block rate, but lowers the resistance against tampering. Therefore, external agreements between all stakeholders are established to reach the desired trustworthiness. According to Hampton, omitting the PoW in a private environment would be no problem. The author argued that no majority attack is needed to tamper a private blockchain, since 100 Percent of all nodes are controlled by one stakeholder [58]. Consequently, attackers who could control all nodes are already controlling the whole private network [58]. A typical feature of some private blockchains is the native capability of storing larger data sets; therefore, private data can be stored directly without being accessible by others. [31]

2.4.6 Storage Capabilities

OP_RETURN Transaction Outputs

The recommended, but also most controversial option for appending arbitrary data to transactions in all classic blockchains, is the use of opcode OP_RETURN in the Script section of UTXOs. If this opcode is executed the script will fail immediately [46, 43]. While normal nodes will ignore the following bytes, specialized nodes can interpret them as intended. Up to version 0.11.0, Bitcoin does allow only 80 bytes of data appended after the opcode [59]. In addition, exactly one UTXO with only OP_RETURN is allowed per transaction, preventing the database from rapidly increasing due to larger UTXOs. However, this does have a huge drawback in terms of huge data sets like in provenance.

By placing OP_RETURN first, the PubKey Script cannot include a public key hash, which prevents everyone to claim the ownership. Thus, a UTXO is a provable and not spendable transaction output (Listing 2.2) [41]. Subsequently, the amount of Bitcoin from the included input transactions cannot be spent to an address within this UTXO. Instead, the miner will take all Bitcoins from the inputs as mining fee. When using multi-output transactions, a fraction of Bitcoins can be transferred to other addresses, while raising the general fee due to the bigger transaction [41, 32].

```
1   scriptSig:    <sig> <pubKey>
2   scriptPubKey: OP_RETURN {arbitrary data}
```

Listing 2.2: Example script with OP_RETRUN [43]

Further on, the blockchain guide mentions the possibility for nodes to prune these type of UTXO from the databases [32]. As normal Bitcoin full nodes currently do not attempt to do this, alternative nodes may decide to do so in the future [46]; this can lead to the loss of all copies of transaction outputs holding provenance data. But self hosted full nodes, which do not drop these entries, can partially solve this problem. Lastly, the PubKey Script field is not encrypted [43]. To protect private provenance data, strong cryptography is mandatory. Depending on the encryption method this will lower the space for arbitrary information, too. One advantage is the strong proof of existence provided by the block, which includes a transaction that yields this type of UTXO. After the block was added to the chain and more blocks were mined on top, it becomes statistically unlikely that the transaction will be removed in the future. Therefore, the

current version of the Bitcoin blockchain provides a very strong tamper-resistance for provenance data, since all transactions including the UTXOs are stored on multiple nodes and validated by them.

Coinbase transaction

Another approach to store data in blockchains and altcoins is the coinbase transaction, which is added to the block by a mining node [32]. In contrast to normal transaction, a coinbase transaction does not have a predecessor to whom it could reference to in the transaction input [60, 46, 41]. Therefore, the SignatureScript does not need to hold a public key and signature to prove the ownership [60, 46]. The 100 bytes available are only limited by the block height, which must be placed in the first four bytes of the Script. Therefore, the Script section can hold up to 96 bytes of arbitrary data. One part that may limit the bytes available is the extra nonce, which allows full nodes to extend the four byte nonce from the block header by 8 more bytes. This quirk was introduced to raise the probability for finding a valid block header, which is very unlikely within the provided range of the standard nonce [60].

A major drawback of this approach is the need of a mining node in order to create valid blocks, which will be accepted from the network. Adding a block to the chain is very unlikely without having proper mining capabilities. But by considering the problem of pruned UTXOs in normal transaction, a self hosted node should be used for ensuring access to all stored data over longer time periods. Therefore, coinbase transactions offer the opportunity to add additional space for provenance data and, subsequently, coins gained from mining a block can be used as input for further transactions. These are then directly linked to the coinbase transaction, which provides the possibility to link additional related information between transactions. Since coinbase transactions will never get pruned from any database on full nodes, they can be considered as strong tamper-resistant and very reliable [37, Ch.8].

2.5 Alternative Blockchains and Blockchain-like Databases

In contrast to Altcoins, alternative blockchains and blockchain-like databases are independent from the Bitcoin implementation. Most of these chains are developed with a different focus, such as more efficient storage of data and assets or building a more general blockchain concept [31]. Two alternatives are explained in the next section, to determine their capabilities in terms of data storage and tamper-resistance. As in the previous section, a shorter overview can be found in the appendix (Table B.1).

2.5.1 Ethereum

The Ethereum blockchain was first proposed in 2013 and initially released in May 2015 as open source project by Buterin [61]. It focused on generalising the blockchain to provide a base for more complex application than distributed ledgers in digital currencies [62, 63, 64, 65]. According to its Authors,

> "[...] Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions [...] [61]".

Ethereum can be used in private or public networks due to its available source code [66]. While having is own currency - Ether - and using similar concepts in terms of networking, mining and PoW, Ethereum tries to overcome some major drawbacks in Bitcoin and other closely linked blockchains implementations. According to Buterin these are, "Lack of Turing-completness",

"Value-blindness", "Lack of State", and "Blockchains-blindness" [61]. The "Lack of Turing-completeness" refers to the fact that the script language of Bitcoin cannot handle loops due to security reasons, which is very space inefficient in some cases. "Value Blindness" refers to the concept of changing the output on behalf of external parameters, when the UTXO is used. For example the current exchange rate, on a specific date in the future. Since Bitcoin only handles the binary state 'spent/unspent' for UTXOs, this is referred to as "Lack of State". Ethereum instead is capable of handling multiple states, which makes the developing of meta protocols and stateful contracts easier. "Blockchains-blindness" refers to the fact that UTXOs in Bitcoin cannot access blockchain values such as block headers. But according to Buterin, this might be a useful resource of randomness for contracts and applications in Ethereum [61]. These ideas later take place in form of definitions to the Ethereum Virtual Machine (EVM) in the paper *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER* by Wood, a co-author of Ethereum [67].

According to this work, the state in Ethereum is represented by accounts rather than by UTXOs as in Bitcoin. These accounts are reachable by a 20 byte address. To maintain the state each account holds: its own storage, its current balance in Ether, a transaction-counter (nonce), and the contract code. There are two types of accounts in Ethereum. The first one is owned by external users and controlled by private keys, similar to Bitcoin. The second type is created internally and controlled by the contract code itself. Internal accounts are capable of creating new internal accounts during execution. The latter is not possible in Bitcoin, but necessary for allowing multiple state transitions.

Ethereum uses messages between accounts to trigger state transitions, these messages are similiar to transaction in Bitcoin. They include the address of the receiving account, the amount of Ether to transfer, some data to transmit, and values for STARTGAS and GASPRICE. Mining nodes which include messages into a block, must execute the contract of the destination account. While executing, the GAS balance is reduced by a specific fee depending on the opcode's used in the contract. This mechanism prevents endless loops, which can occur during the execution of a transaction due to the Halting problem.

Due to its focus, the block rate in Ethereum is about 50 times faster than in Bitcoin [67, 65]. As in other blockchains, this allows for faster processing of transactions but also for a faster execution of the code. With regards to a similar foundation to the generic blockchains and the size of the network on the one hand, Ethereum is as tamper-resistant as Bitcoin. But the Turing-completeness of Ethereum can lead to attacks based upon exploitable contract code, as evidenced in the Decentralized Autonomous Organization (DAO) incident [68] Consequently, the Ethereum blockchain was hard forked by a new versions of the blockchain software. Mining nodes which agreed with this release started to mine on a previous block before the incident. Since the Ethereum project changed some of the consensus rules in favour of its biggest stakeholder, the predicate of reliability and tamper-resistance was undermined. Subsequently, the project was split into two projects, were both are mining on separate forks.

Storage Capabilities

Like in Bitcoin, the Ethereum community mostly dislikes approaches utilizing the blockchain for storage of huge data sets. But in contrast to other chains, Ethereum is explicitly designed to store data using two different methods. One utilizes the storage for data in accounts the other uses events also known as Log storage. Both methods are implemented within the contract code. For better explanation of the following two examples, Solidity as high level language for contract description is used [69].

Storage As briefly outlined before, every account in Ethereum has extra storage for data. This storage is implemented in the EVM as key-value store with 256 bit word size [67]. It can be read out, or written during the execution of the contract and interpreted as boolean, integer or byte arrays. As shown on line 2, Solidity uses variables to accesses the storage, which in this case is a 256 bit unsigned integer (Listing 2.3) [69]. In addition, Solidity offers modifiers which add the ability to secure variables from being written by others than the owner. This functionality is shown in line 5 and used to protect the set function in line 7. After compiling the script to EVM code, the contract must be created by an external account by sending a special message [67]. As a result, a new addressable account exists in Ethereum where other accounts can read out the current value by sending a message but only the owner account can write it.

```
1 contract SimpleStorage {
2   uint private storedData;
3   address owner = msg.sender;
4
5   modifier onlyOwner { if (msg.sender != owner) throw;_}
6
7   function set(uint x) onlyOwner { storedData = x; }
8   function get() constant returns (uint) { return storedData; }
9 }
```

Listing 2.3: Solidity Script for storing Data in contracts [70, 71]

This technique has the advantage of being intuitive and easy to use for programmers. One disadvantage is that the use of higher level scripting languages hides the complexity of the resulting EVM code. Since the fee for the execution of a contract is calculated on the executed operations in the EVM, the resulting fees highly depends on this code. This is especially true for operations like SSTORE, which cost 20000 GAS each execution. Since 32 bytes are written in one call to the operation, 1024 byte of data will consume about 640000 GAS. At a current GAS price of 18GWei, this accumulates to 0.01152 Ether or approximately 0.13 € per kByte [72]. This calculation does not include additional costs for setting up a message and other operations involved in executing the contract. Therefore, a real contract would be much more expensive. Another advantage is the possibility to write contracts with more complex methods, which may enable rich querying of data. On the downside, the added complexity can lead to unintended changes to the data in storage. One major disadvantage, in terms of tamper-resistance and reliability, is the possibility to destruct accounts by sending a prepared message from an external owner account. In consequence, the storage went inaccessible. In the current version of Ethereum, the history of deleted accounts will not be dropped from the database, but this may be possible in the future [63]. Similar to other blockchains, data that is written to the contract storage can be read out by everyone who stores the data in its copy of the blockchain [67].

```
1 contract ClientReceipt {
2   event Deposit(address indexed _from, bytes32 indexed _id, uint
       _value);
3   function deposit(bytes32 _id) {
4     Deposit(msg.sender, _id, msg.value);
5   }
6 }
```

Listing 2.4: Solidity Script for simple Storage using Events [73, 70]

The second approach to store data in Ethereum is the use of the log storage, also known as events [63, 67]. Its intended use is to send return values or logging events to a specific account. As depicted on line 3, sending a message which calls the deposit function to an account holding the contract, will cause a subsequent message returned to the sender (Listing 2.4) [69]. This message will include the value associated with id given in the received message.

One disadvantage compared to the storage approach is that data once stored in the log can only be accessed from external accounts, or application with access to the blockchain data structure [63]. Subsequently, later queries are more difficult to handle than in the storage solution. One advantage is that all messages are stored in the transaction database. Therefore, they will never get dropped from the blockchain and can be considered more reliable and tamper-resistant. In addition, the information about successful storage operation may be useful in some applications. Another advantage is that log storage is less expensive than normal storage. Storing 1 kByte of data in the given example will consume 8942 GAS, which accumulates to 0.000160956 Ether or 0.0019 € [72]. As in the previous example, this does not include additional fees.

	Traditional Blockchain	Traditional Distributed DB	BigchainDB
High Throughput; increases with nodes↑	-	✓	✓
Low Latency	-	✓	✓
High Capacity; increases with nodes↑	-	✓	✓
Rich querying	-	✓	✓
Rich permissioning	-	✓	✓
Decentralized control	✓	-	✓
Immutability	✓	-	✓
Creation & movement of digital assets	✓	-	✓
Event chain structure	Merkle Tree	-	Hash Chain

Figure 2.6: Properties of BigchainDB [74]

2.5.2 BigchainDB

BigchainDB was founded by Ascribe in 2016, following the goal to provide owners of digital assets with a system that guarantees a reliable proof of ownership in terms of copyright and licensing [75]. Since BigchainDB is open source software it can be used in private and public networks. BigchainDB focusses on a fast and query rich blockchain-like database that is capable of storing huge amounts of assets, but also includes features that are attributed to blockchains (Figure 2.6) [74]. Consequently, BigchainDB utilizes the chaining concept of transactions and time-stamping of blocks from blockchains. In contrast to other blockchains, BigchainDB does not implement its own peer-to-peer network. Instead, distributed databases like RethinkDB and MongoDB are used to relay transactions and blocks to other nodes. A node either connects to an existing database cluster or sets up its own database, which in turn connects to other instances. Similar to Bitcoin new transactions are validated by nodes and collected into blocks, which are written into the blockchain table. These blocks are linked together by the hash of the previous block. Instead of securing the blockchain by PoW, each node votes on each block by signing it with its digital signature (2.5). A node signs a block as valid, if each transaction in that block is valid, based on the consensus rules of the nodes. According to the development reference the validation process can be changed by applying a new set of rules by using plugins [76].

Like in Bitcoin, a user can easily create a new cryptographic key pair which identifies his account. In contrast to other blockchains, accounts connected to BigchainDB nodes cannot receive messages about votes on blocks due to the handling in the database backend [74, 76]. This concept is different from generic blockchains, were each client receives messages about new valid blocks from the network. Therefore, accounts must trust in the nodes they are connected to without being able to prove the valid state of the blockchain by themselves. Since BigchainDB focus on assets as valuable content of the chain, accounts can issue a special transaction, which starts a new chain of transactions by introducing a new asset. This concept replaces the generation of new coins in the PoW of currency-based blockchains.

Similar to other blockchains, accounts can prove the ownership of assets by signing the appropriated transactions outputs with their private key. If an account wants to transfer the ownership to another account a new transaction must be created and send to a node.

In its current beta state, BigchainDB must be stated less tamper-resistant compared to other blockchains, because of the concept of handling blocks and connecting clients. Two other major

```
{
    "block": {
        "node_pubkey": "JCPMWqz8QB6PfkeKn8xSVG2NPDu4qEgR8iBAZM2rgnPk",
        "timestamp": "1489771157",
        "transactions": [],
        "voters": ["JCPMWqz8QB6PfkeKn8xSVG2NPDu4qEgR8iBAZM2rgnPk"]
    },
    "id":
↪   "515243eab600e6415eb414d7a3707bebf849465d2f0e1060d5ff3e1e2512e239",
    "signature":
↪   "3xZJZBgPYSzuZqdMKQZjsmqUDfQZPmuwksAE3DUt2M7t1dSfRKJiK9..."
}
```

Listing 2.5: Basic block data structure in BigchainDB

facts also adding to this conclusion. First, the public peer-to-peer network of nodes is smaller compared to other networks. Second, external databases does allow access by administrative users, who can read and write to the main database. By dropping all tables this may result in a complete loss of all data. However, due to the chaining and signing of transactions and blocks an alternation of single entries is still very unlikely [74].

Storage Capabilities

As explained above, transactions in BigchainDB can hold information about assets they describe. According to the developer documentation a transaction does include an explicit field *asset* to store information about it [76]. The field *operation* contains a string indicating the transaction type (2.6). These types are CREATE, TRANSFER, and GENESIS. In case of a CREATE transaction, the field *asset* can hold a dictionary with the key *data* and an arbitrary JavaScript Object Notation (JSON) document as value. Consequently, this can be utilized to store data.

If the asset is transferred to another account, the *id* of the transaction holding the asset data is placed in the value of the *asset* field. In order to indicate the transaction as ownership transfer, the field *operation* must be filled with the value TRANSFER. In addition, of putting data into the asset, data can be attached to the *metadata* field for encoding information related to the transaction itself. Therefore, the metadata can hold arbitrary in any transaction type.

Since BigchainDB does not include a currency, one advantage is that transactions do not generate any extra costs [74]. In addition, the asset or data size is not limited in any way. One disadvantage is the large network overhead due to HyperText Transfer Protocol (HTTP), which is used for communication with the node [76].

```
{
        "id": "<hash of transaction, excluding signatures>",
        "version": "<version number of the transaction model>",
        "inputs": ["<list of inputs>"],
        "outputs": ["<list of outputs>"],
        "operation": "<string>",
        "asset": "<digital asset description>",
        "metadata": "<any JSON document>"
}
```

Listing 2.6: CREATE Transaction in BigchainDB [76]

Chapter 3

Related Work

To the best knowledge of the author, there is as of yet no direct approach to store provenance data in blockchains or blockchain-like databases. This is especially true when encoded with the PROV standard. Other works have used the blockchains to store native assets or metadata. More recently, studies have utilized the time-stamping of blocks to generate provenance about documents and provide better tamper-resistance for them.

Representatives of the first use case are Open Assets and Colored-Coins, which use the OP_RETURN opcode to append data [77, 78]. Since the amount of space is limited, both approaches depend on binary sub-protocols. They are intended for storing a derived hash of an external document along with a quantity and metadata. The ownership transfer is represented by a transaction to the address of a new owner. The more complex protocol of Colored Coins also utilizes multi-signature addresses to associate multiple transactions, including unspendable UTXOs with one asset. Consequently, the protocol allows for storing more than 80 Bytes.

A similar concept for digital assets, which aims on securing legals and controlling intellectual property was proposed by Ascribe in the work *Towards An Ownership Layer for the Internet* in June 2015 [79]. As stated in the white paper, Ascribe mentions the possibility of tracking assets due to the implicit provenance the blockchain provides. In contrast to other concepts, Ascribe describes the concept of using a unique owner address as id for derived multiple editions of an asset.

Another approach was proposed by Irving and Holden in the article "How blockchain-timestamped protocols could improve the trustworthiness of medical science [version 2; referees: 3 approved]" to improve the trustworthiness of data in medical science [80]. It describes the possibility of hashing data or a document into a SHA256 digest, which in turn is used to create a key-pair. Subsequently, a Bitcoin address is derived from the public key and used to set up a single transaction, which serves as "proof-of-existence". However, as mentioned by the author, anyone who owns the original document can easily recalculate the key-pair and claiming the Bitcoin address. A major disadvantage of all approaches is the lack of storage capabilities for larger data sets. But the development of a dedicated protocol may offer further possibilities for storing provenance natively in the blockchain.

A more complex concept was proposed by Nugent, Upton, and Cimpoesu in the article "Improving data transparency in clinical trials using blockchain smart contracts [version 1; referees: 1 approved]" [81]. By setting up contracts in a private hosted Ethereum blockchain, the author describes a system which manages all related data from subjects of clinical trials. The system consists of several accounts which are generated with the contract code from a central account. Subjects and subsequent data is added or retrieved by calling special functions on the trial contract from other generated accounts. Larger documents are stored in a InterPlanetary File System (IPFS) and referenced in the corresponding data set. By using a private Ethereum

blockchain, the author showed the capabilities of more complex systems which require additional logic and permissions. Since all metadata about trials is stored in the blockchain the system provides a strong proof-of-existence but also a complex trial history.

Chapter 4

Storage Concepts

4.1 Introduction

As discussed in the previous chapters, blockchains do offer possibilities to store data. The following three concepts were designed to be applicable to all observed currency-based blockchains, and alternative chains, like Ethereum or BigchainDB. Therefore, features like multi-signature transaction were not used in these concepts. Instead, mapping provenance information represented in PROV records to multiple transactions were focused on. Respectively, linking information between transactions was the second major focus of the concepts. In addition, the ownership concept and the possibility of ownership transfers provided by blockchains were included in the concepts. In order to allow more flexibility in each concept, the assumption was made that each account has direct access to all transactions and blocks within the blockchain network. This does not necessarily mean that accounts must run a dedicated full node, but at least access to all past transactions within the used address space is guaranteed.

Every concept is based upon different views on Provenance in its PROV representation. Since the PROV standard offers serialisation of provenance data in formats like PROV-XML or PROV-JSON, many related applications are able to import or export these documents. Exchanging provenance in this way implies a document-based view on the data where each serialisation represents a valid set of provenance records about a given subject. Therefore, the first concept focuses on how to store these serialized documents directly into blockchains. As explained above, most currency-based blockchain have limited space to store data. As a result, the second and third concepts focus on lowering the space needed to store provenance documents by scatter the data across multiple transactions. Since the main model of PROV was designed around types which are linked together by relations, another representation of such provenance is a graph. As such, the second concepts utilises this view on provenance to split up the document. Another common view on provenance data in PROV-DM is the agent-centred view [20]. The relations which represent the responsibility of agents on other types can be used to split up provenance records accordingly; the third concept is designed using this initial idea.

4.2 Document-based Concept

Storing As previously outlined, the goal of the document-based concept is to avoid previous transformation or splitting of the provenance document. No direct linking between transactions is needed thus, all information is included in a single transaction. Consequently, transactions with attached provenance data are created using a single account in the blockchain (Figure 4.1).

The algorithm illustrates that all records form a given provenance documents need to be serialized into an asset which can be stored into the selected blockchain (Algorithm 1, Line 2).

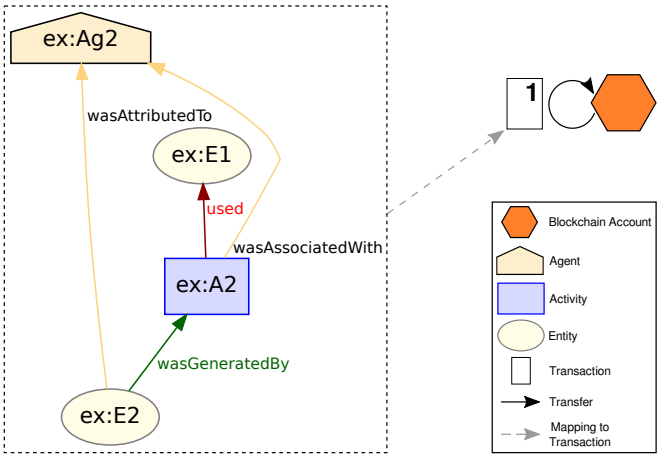


Figure 4.1: Document-based concept - A document is stored in a single transaction

Afterwards a new transaction including the asset is created and signed by the account (Line 3). In order to achieve compatibility with all observed blockchains and to improve the proof of ownership and tamper-resistance, the transaction should be transferred to the same account (Line 4). After waiting for the transaction to get incorporated into a valid block, the transaction id is returned for later use or query's (Line 5-8).

Input: prov_records, account
Output: transaction_id
1 **begin**
2 asset ← serialise_records(prov_records);
3 fulfilled_transaction ← create_and_fulfil_transaction(account.private_key, asset));
4 transaction ← transfer_transaction(account.public_key, fulfilled_transaction);
5 **repeat**
6 status ← get_status(transaction.id);
7 **until** *status = "valid"*;
8 **return** transaction.id;
9 **end**
Algorithm 1: Write PROV records into the blockchain using the document-based concept

Querying To query documents form the blockchain one account must receive the transaction by its id (Algorithm 2, Line 2). In case of ownership transfers, further transactions following the chain must be requested until the transaction with the asset is received. Thereafter, the asset is deserialized into the original provenance records and returned (Line 3-4).

Input: transaction_id
Output: prov_records
1 **begin**
2 transaction ← get_transaction(transaction_id);
3 prov_records ← deserialise_records(transaction.asset);
4 **return** prov_records;
5 **end**
Algorithm 2: Get PROV records from blockchain using the document-based concept

Concept properties The major advantage of the document-based concept is its simplicity. This is due to a single account that must interact with the blockchain. By knowing the used account and owning all of its transactions, querying of documents is very efficient and easy to implement. However, storing PROV documents with addresses associated to a single account also simplifies potential attacks. Due to this single point of failure and the fact that a document is stored within one transaction, the concept should be considered less tamper-resistant and reliable. Additionally, a huge payload per transaction leads to further problems in currency-based blockchains. Firstly the fee to pay to successfully include a transaction into a block will proportionately increase; secondly, the usual limititation in byte size does not allow storing of big data sets. Therefore, complexity must be added in order to save space. In this case a similar approach as in Open Assets can be utilized if a blockchain like Bitcoin is used. The last problem in particular, implies the use of alternative blockchains for the document-based concept. In particular, BigchainDB could be used to create a file store with blockchain capabilities.

Use Cases The concept can be used to implement a simple document store for independent users. On these grounds, the concept is similar to the ProvStore, which was proposed and implemented by Huynh and Moreau [82]. Another use case for the concept might be transparent Application Programming Interface (API) for automated storage of PROV documents by multiple components of more complex systems. In both cases a single account can be used for each user or component, in order to manage the documents and provide faster access.

4.3 Graph-based Concept

Storing Instead of store provenance records in one transaction, the graph-based concept utilizes ownership transfers to represent PROV relations between all types within the PROV-DM. Therefore, each agent, activity or entity must be represented with an account in order to create new transactions and initialise ownership transfers. To find all types in the provenance records from the input, the document is split up into three sets (Algorithm 3, Line 4-5). These are a set with types, and sets including relations with and without identifiers as the first attribute. Next, the accounts are created or otherwise queried from a local database, using the set of types (Line 7-14). Subsequently, each account serializes the record to an asset (7-14). As implied in the transactions 1 to 4, this is the declaration of the types and their attributes (Figure 4.3). The serialized provenance data is then incorporated into a new transaction and announced to the blockchain network to get incorporated into a block. To quickly determine the ownership, the transaction is sent and received by the same account, as depicted in the mapping figure below (Figure 4.2).

After this step, each account must create all relations from the sets containing them (Line 19-29). Therefore, each relation which is related to the type the account stand for, is taken from the sets. Subsequently, the receiving account is inferred from the relation itself. After serialising the PROV records to an asset it is included in a new transaction signed, and transferred to the other account (Figure 4.2). For example, the `used` relation at activity A2 is transferred in transaction 7 to entity E1 (Figure 4.4). Since the activity is already described in the initial transaction and the ownership is clear due to address of the account, the information can be left out in the attached data. For the same reason no further description about the entity is needed, since the transfer can be used to infer this information from the blockchain. To query additional provenance about the involved types faster, their transactions ids can be included in the mapping inside of the transaction. Lastly, all transaction ids collected form total transfers are returned for later query's.

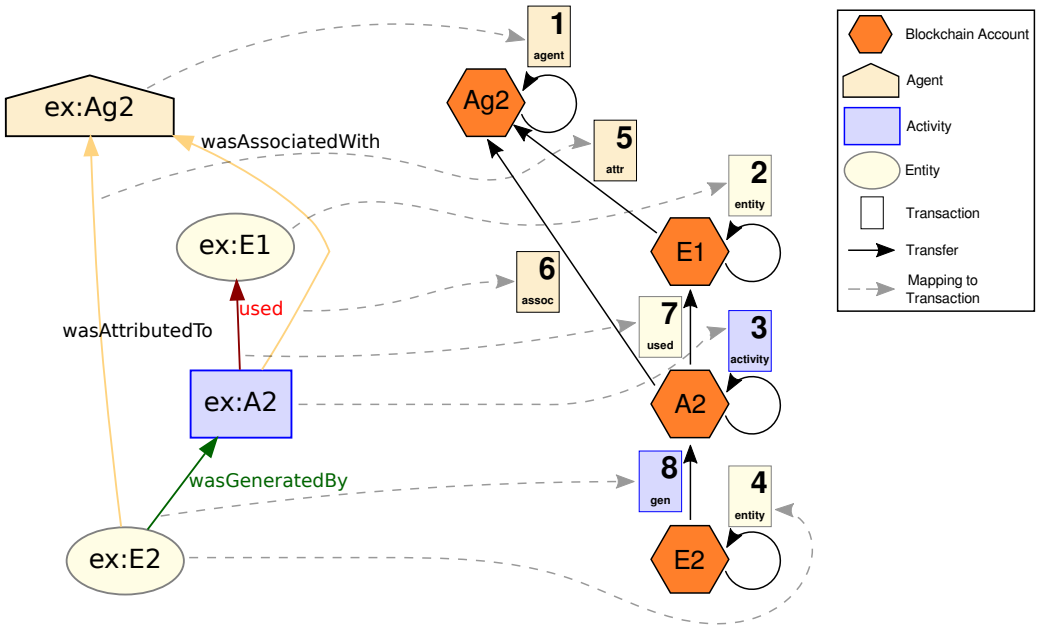


Figure 4.2: Graph-based concept - Transactions sent between blockchain accounts. Coloured in ownership after successful transfer.

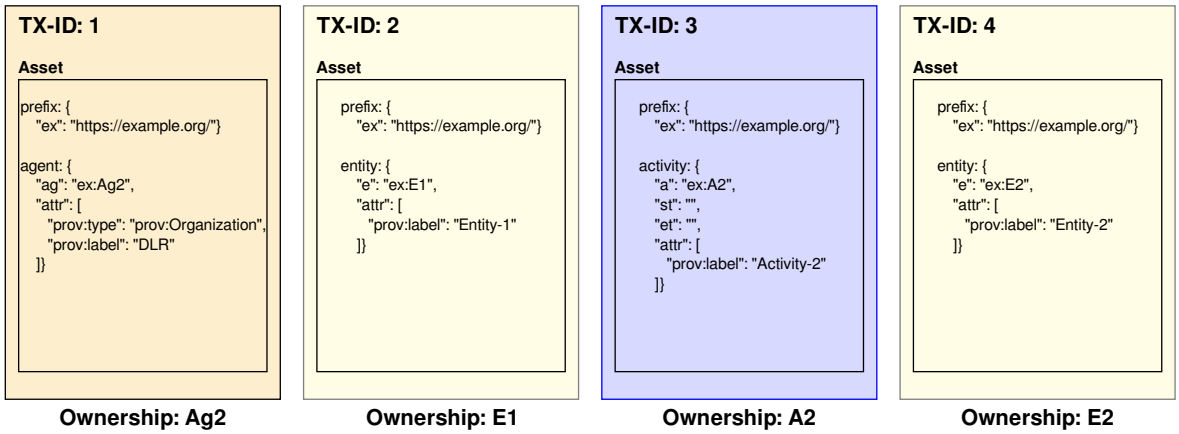


Figure 4.3: Graph-based concept - Transactions, describing the types

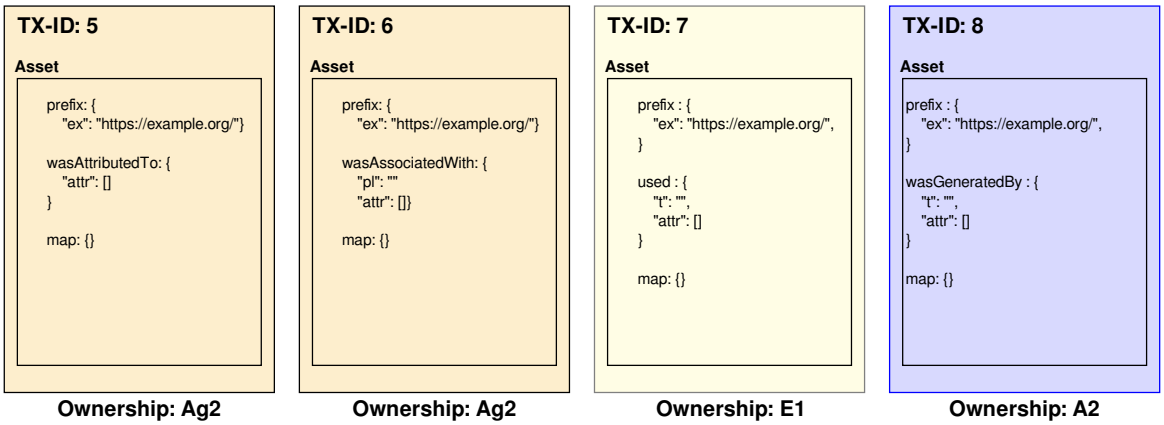


Figure 4.4: Graph-based concept - Transactions, describing relations between types

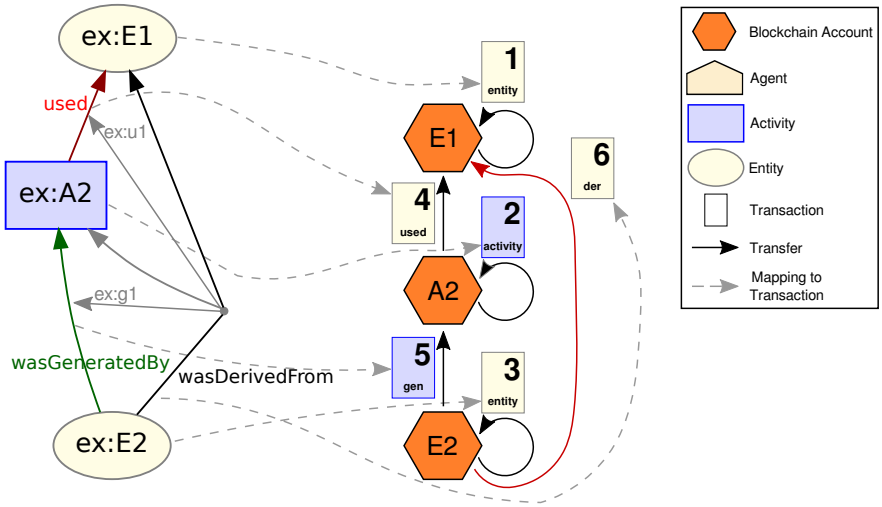


Figure 4.5: Graph-based concept - Ids used in wasDerivedFrom() forces transactions to be in specific order

The concept so far does not allow a complete description of relations like `wasDerivedFrom(id; e2, e1, a, g1, u1, [attr])` or `wasAssociatedWith(id; a, ag, pl, [attrs])`, because of references to additional types and relations in the arguments (Figure 4.5). Thus, a mapping between the referenced relations and types to their corresponding transaction ids must be included into the transactions. Therefore, all transactions which describe relations with a defined identifier must be created first to ensure valid mappings. As shown in the figure, this means that the transactions 4 and 5 with the identifiers ex:g1 and ex:u1 must be prior to transaction 6 (Figure 4.5). In case of the example the transaction ids of the referenced relations g1 and u1 as well as the activity A1 are needed to set up a valid transaction for the wasDerivedFrom relation.

```
Input: prov_records
Output: transaction_ids
1  begin
2      transactions_ids ← {};
3      accounts ← {};
4      elements ← {record ∈ prov_records|type(record) = prov_element};
5      all_relations_with_id ←
         {record ∈ prov_records|type(record) = prov_relation ∧ has_id(record) = True};
6      all_relations_without_id ←
         {record ∈ prov_records|type(record) = prov_relation ∧ has_id(record) = False};
7      foreach element in elements do
8          account ← create_or_reuse_account(element);
9          if !account.txid then
10             transaction_id ← create_asset(account.private_key, account.public_key,
                 element);
11             account.txid ← transaction_id;
12         end
13         transactions_ids ∪ {account.txid};
14         accounts ∪ {account};
15     end
16     foreach account in accounts do
17         relations_with_id ←
             {relation ∈ all_relationss_with_id|is_out_edge(relation, account) = True};
18         foreach relation in relations_with_id do
19             recipient ← get_recipient_account(relation);
20             transaction_id ← create_asset(account.private_key, recipient.public_key,
                 relation);
21             transactions_ids ∪ {transaction_id};
22         end
23     end
24     foreach account in accounts do
25         relations_without_id ←
             {relation ∈ all_relations_without_id|is_out_edge(relation, account) = True};
26         foreach relation in relations_without_id do
27             recipient ← get_recipient_account(relation);
28             transaction_id ← create_asset(account.private_key, recipient.public_key,
                 relation);
29             transactions_ids ∪ {transaction_id};
30         end
31     end
32     return transactions_ids;
33 end
```

Algorithm 3: Write PROV records into the blockchain using the graph-based concept

```
Input: element, private_key, public_key
Output: transaction_id
Function create_asset(element, private_key, public_key):
    asset ← serialise_records(element);
    fulfilled_transaction ← create_and_fulfil_transaction(private_key, asset));
    transaction ← transfer_transaction(public_key, fulfilled_transaction);
    repeat
        status ← get_status(transaction.id);
    until status = "valid";
    return transaction.id;
```

Algorithm 4: Algorithm for create_asset() function

Querying To query data from the blockchain the desired data must be requested by the transaction id. By knowing all accounts, transactions can be directly requested from the blockchain using their ids (Algorithm 5). By having prior knowledge it is possible to rebuild the complete graph by first creating all records of all involved types, using the initial or first transaction of each accounts. Afterwards all transactions with relations are used to set up the links between the types.

```
Input: transaction_ids
Output: prov_records
begin
    prov_records ← {};
    foreach transaction_id in transaction_ids do
        transaction ← get_transaction(transaction_id);
        prov_record ← deserialise_records(transaction.asset);
        prov_records ∪ {prov_record};
    end
    return prov_records;
end
```

Algorithm 5: Get PROV records from blockchain using the graph-based concept

Concept properties The major advantage of the graph-based concept is the large amount of transactions needed to build the graph, which guarantees a difficult to tamper data structure. A potential attacker must double-spend multiple transactions i*n order to* successfully alter the provenance data. Since multiple addresses are involved in building the graph, double-spending becomes even more unlikely. Compared to the document-based concept another advantage is the smaller payload size in each transaction, which potentially allows operators to use the concept in currency-based blockchains. The concept also offers easy querying of information about a particular type, due to its representation as account with a unique address in the blockchain. By querying all owned transactions of one account its is possible to investigate every asset pointing to this account. Therefore, it's easy to determine which entities were created by a specific activity. Another advantage is the possibility that other applications are able to extended the graph as needed. Since each application knows its own address space by knowing all accounts, every transaction from an external address can be considered untrusted; as such, no harmful additions are possible. Querying the graph from the example involves more requests to nodes in the peer-to-peer network due to the distributed information. The fact that multiple addresses are maintaining the whole set of transactions and the processing needed to reassemble the

graph leads to significantly slower query performance, as in the previous approach. Another possible disadvantage is the need for a more complex upper layer, which supplies all addresses with a sufficient amount of coins in order to pay the fees for each transaction in a currency-based blockchain. In addition, the strongly linked data structure of PROV and the resulting amount of transactions will lead to higher costs compared to the document-based concept. The last disadvantage entails the transaction ids which are needed in some PROV relations and for querying data. To provide valid transaction ids for the mappings, a local data structure must be maintained. To guarantee only valid transaction ids as references in transactions the local mapping must be updated if the previous transaction was successfully included into the blockchain. As mentioned, referencing types and relations with transaction ids also require some transactions be transferred before others in order to complete the mapping. Therefore, storing of large provenance graphs will be less performant.

Because of the overall added complexity, the graph-based concept is easier to deploy on alternative blockchains. Especially contracts in Ethereum may help to solve the problem of the required ordering of transactions, due to possible account creation and internal logic. The unlimited storage capacity in Ethereum and BigchainDB also allows additional information to each transaction, which may solve the problem of external references represented by transaction ids. However, compared to generic blockchains the use of alternative blockchains will lower the reliability and tamper-resistance due the missing PoW in BigchainDB and the possibility of contract deletion in Ethereum.

Use Cases One appropriate use case of the graph-based concept are systems, which are more likely to add single relations between types rather than storing documents at once. For example, this is useful for a larger set of applications, which participate in extending the same provenance graph by sharing its address space.

4.4 Role-based Concept

Storing The role-based concept emphasizes the role of agents by directly ascribing the ownership of all related entities and activities to them. Therefore, the PROV records are first split up into three sets (Algorithm 6, Line 5-6). The first set holds every element typed agent in PROV, while the second set holds all other types. The third set holds the relations. Following on line 7 to 18, the accounts are created from the set containing the agents where afterwards an asset is created that first holds all provenance information about the agent. In addition, all out going relations from the agent are filtered from the global set and added to the asset (Figure 4.7). As shown in transaction 1 and 2 the asset is then transferred to the same account (Figure 4.6).

After each account is created, the types with association or attribution to the agent are filtered from the set respectively (Algorithm 6, Line 19-30). For each type an asset is created which holds the provenance information about that type and all out going relations from it; the asset is then transferred to the same account. In the example figures, this is depicted in transactions with ids 3 and 4. Similar to the previous concept, the transaction is then announced into the network using the same account as sender and recipient.

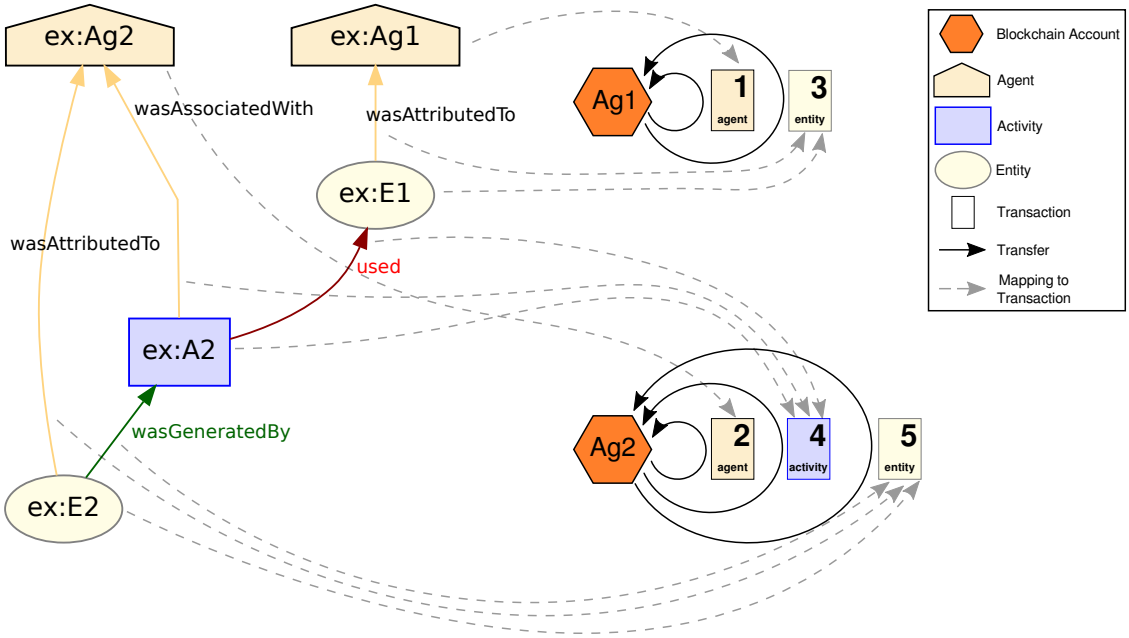


Figure 4.6: Role-based concept - Transferred transactions between accounts

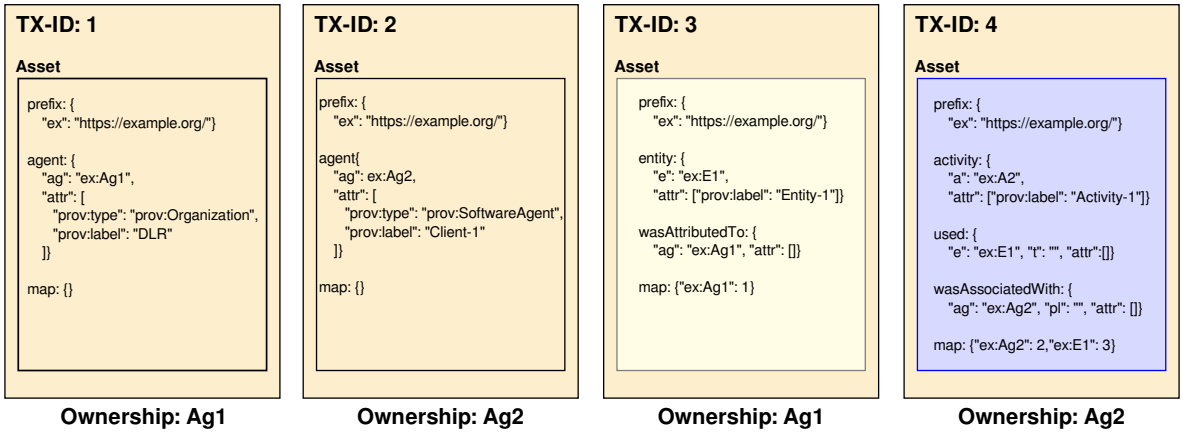


Figure 4.7: Role-based concept - Transactions, describing types with all outgoing relations to other types

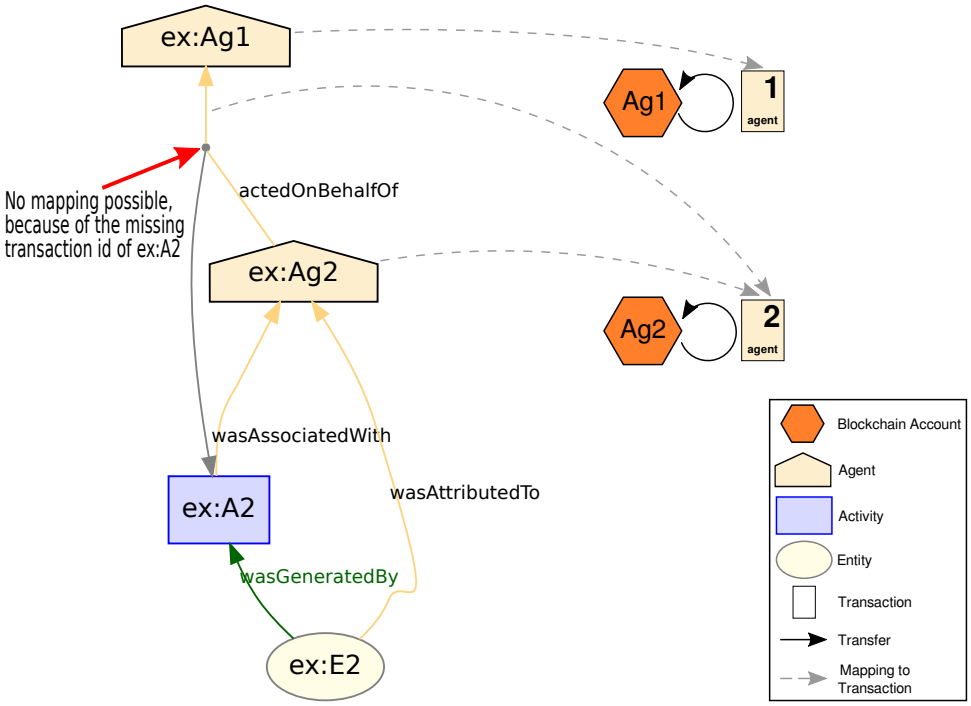


Figure 4.8: Role-based concept - Mapping of Id in actedOnBehlafOf() relations

In order to allow queries by external blockchain accounts, the concept also uses a mapping to reference types and relations in transactions by transaction id. Consequently, the elements must be created in a specific order such as in the graph-based concept. By virtue of the concept design, this leads to a problem regarding the `actedOnBehalfOf` relation, if such a relation incorporates an activity as third attribute (Figure 4.8). Since the concept requires that agents must be created first, it is not possible to reference the activity in the mapping of the `actedOnBehalfOf` relation with its transaction id. In the example this would be the case for the transaction, which creates agent Ag2 who acted on behalf of agent Ag1 but also bears responsibility for activity A1 .

Querying The query of records does work like in the graph-based concept. By requesting all transaction by id, it is possible to rebuild the provenance records using the same algorithm (Algorithm 5, Line 19-30).

Concept properties One advantage of this concept is that all necessary information about specific activities or entities is included in one transaction. By knowing the account related to an agent, defining a query that returns all types and instances one agent was responsible for is simpler. But queries that should include information about instances referenced in the mapping might be slower, since additional transactions must be read from the blockchain. Another advantage is that fewer accounts are needed, such as in the previous concept, to store the same provenance data in the blockchain - this lowers the complexity. Since the average size of transactions is smaller compared to the document-based concept, fees in currency-based blockchains are expected to be smaller for one transaction. In addition, the less complex transaction structure of the concept, compared to graph-based concept, also leads to fewer transaction in general. Being an advantage from the cost perspective, fewer transactions and participants are lowering the reliability and tamper-resistance compared to the previous concept.

Input: prov_records
Output: transaction_ids

```
1  begin
2      transactions_ids ← {};
3      accounts ← {};
4      agents ← {record ∈ prov_records|type(record) = prov_agent};
5      all_elements ←
         {record ∈ prov_records|type(record) = prov_entity ∨ type(record) = prov_activity};
6      all_relations ← {record ∈ prov_records|type(record) = prov_relation};
7      foreach agent in agents do
8          account ← create_or_reuse_account(agent);
9          if !account.txid then
10             asset ← {agent};
11             relations ← {relation ∈ all_relations|in_out_edge(relation, agent) = True};
12             asset ∪ relations;
13             transaction_id ← create_asset(account.private_key, account.public_key, asset);
14             account.txid ← transaction_id;
15         end
16         transactions_ids ∪ {account.txid};
17         accounts ∪ {account};
18     end
19     foreach account in accounts do
20         elements ← {element ∈ all_elements|has_relation_to(element, account) = True};
21         foreach element in elements do
22             asset ← {element};
23             relations ← {relation ∈ all_relations|is_out_edge(relation, element) = True};
24             foreach relation in relations do
25                 asset ∪ {relation};
26             end
27             transaction_id ← create_asset(account.private_key, acount.public_key, asset);
28             transactions_ids ∪ {transaction_id};
29         end
30     end
31     return transactions_ids;
32 end
33 Function create_asset(element, private_key, public_key):
34     asset ← serialise_records(element);
35     fulfilled_transaction ← create_and_fulfil_transaction(private_key, asset));
36     transaction ← transfer_transaction(public_key, fulfilled_transaction);
37     repeat
38         status ← get_status(transaction.id);
39     until status = "valid";
40     return transaction.id;
```

Algorithm 6: Write PROV records into the blockchain using the role-based concept

Another disadvantage is that the concept is not intended for use with provenance data that does not include agents. This may lead to misinterpreted or even invalid provenance data for other blockchain accounts, which expect a valid PROV agent behind every address. Another pitfall related to the same problem is based upon the PROV standard itself, which allows multiple agents to be attributed or associated with entities or activities. Subsequently, two transactions including the same type and relations are created if two agents are attributed or associated with the same type. The problem can be avoided with two possible solutions. First, the implementation defines a rule to determine the agent who is responsible for creation of the transaction. Second, the implementation is able to merge duplicated information from different transactions. As a result of the last two problems, the concept is only applicable in specialised environments with well defined provenance concept.

Due to its larger transaction size compared to the graph-based concept, the concept does not allow the use of generic blockchains without additional optimization on the payload. Therefore, the role-based concept is a candidate for alternative blockchains like Ethereum and BigchainDB, taking similar problems in terms of complexity compared to the graph-based concept into account.

Use Cases Due to the agent centric view, one usage example might be a distributed system with a set of workers, which work on behalf of external agents. By collecting provenance about the execution of tasks and related data in a process, such systems could provide helpful information for understanding the reasons of a failed process. Another example are software applications, which collect provenance about activities and entities from independent users or agents. Storing the provenance with the role-base concept would then allow for easier queries of specific activity or entities within the responsibility of a particular agent.

4.5 Concept Comparison

As explained above, each concept has varying advantages and disadvantages in terms of tamper-resistance, payload size, flexibility and their complexity in storing and querying. Regarding the focus on trustworthy provenance in this work, the graph-based concept is the most appropriate for achieving this goal. As depicted it presents the best tamper-resistance and reliability of all concepts (Table 4.1). By having the smallest payload size, implementations in all blockchains and altchains mentioned in the background chapter are possible. In addition, the flexibility yield by the graph-based concept allows a usage in wide variety of applications. In particular, the possibility of independently extending the graph with multiple applications could be useful if information should be shared between several stakeholders. The drawback is the complexity needed to store and query provenance data. Therefore, the concept is not applicable when fast storage or querying is required. The requirement could be adhered to with the document-based concept by accepting the downside of a less tamper-resistant storage. This is especially true if alternative blockchains are used in order to compensate the overall bigger payload size. In terms of the observed parameters, the role-based concept does offer average results. The concept is more likely to be used in specialised system with a strong provenance concept, due to its complexity in storing new provenance data and the strong focus on agents. Provenance data with multiple attributions or associations to one instance need special treatment in order to go along with the constraints of the PROV-DM.

Table 4.1: Concept Comparison

	Document-based	**Graph-based**	**Role-based**
Storing	+	-	-
Querying	+	-	o
Write Performance	+	- -	o
Payload Size per TX	-	+	o
Usage flexibility	-	+	o
PROV compatibility	+	+	-
Tamper-Resistance	-	+	o
Reliability	-	+	o

Chapter 5

Software Prototype

5.1 Requirements and Definitions

To demonstrate and test the capabilities of the proposed concepts a software prototype was developed. In order to define a suitable concept, a set of requirements and definitions were constituted. In terms of the general software prototype the following requirements were defined:

P.1 The prototype implements all proposed concepts.

P.2 The prototype accepts provenance data in PROV-JSON and PROV-XML.

P.3 The prototype offers the same API for each concept.

P.4 The prototype is modular extendable by additional concepts.

To have a common base in functionality and allow comparison of the concept in the analysis, the following requirements were made for each concept:

C.1 Each concept ensures that all transactions to the blockchain are incorporated into blocks.

C.2 Each concept is capable to retrieve stored documents.

C.3 Each concept ensures that retrieved documents are only rebuilt from valid transactions.

C.4 Each concept is independently testable in terms of performance and tamper-resistance.

First a decision on the used blockchain application was made in order to fulfil the requirement P.1 based upon the following reasons. According to the previous sections, public currency-based blockchains are offering the best tamper-resistance to protect provenance data from changes but, due to their limited payload size and the subsequent need to invent a meta protocol for provenance, the proposed concepts are difficult to apply. Additionally, fees in these blockchains are strongly bound to the transaction size which would raise the costs of storing provenance information in publicly running examples. Furthermore, the fee system requires additional logic to ensure a constant coin supply for each account. The same problems also occur in Ethereum blockchains. Since fees in Ethereum are dependent on transaction size and executed contract code, and new contracts can be created by other contracts, an even more complex logic is required to ensure the coin supply would be needed. Therefore, the blockchain-like database BigchainDB in version 0.9.1 was selected for use in the prototype. This will lead to a reduced tamper-resistance, which can be compensated by running the prototype in a private environment. However, by having a strong focus on storing and querying assets, BigchainDB fits the needs for

handling even large data sets with provenance information. In addition, the blockchain database can be accessed by a Representational State Transfer (ReST) service, which offers functionalities that are similar to APIs in other blockchains. Consequently, the implemented prototype will be close to implementation in other blockchains.

Since BigchainDB only provides a python driver compatible with Python 3.5 and to avoid the overhead of developing an interface for accessing BigchainDB, the decision was made to implement the prototype in Python. By offering methods to create, transfer, requests and validate transactions, the driver can help to fulfil the requirements C.1 to C.3. Due to the ReST service of BigchainDB and the data format used in RethinkDB, the driver is also capable of serializing python dictionaries into valid JSON. Therefore, it's possible to directly attach provenance data in its PROV-JSON representation to assets in transactions.

In order to fulfil the requirement P.2 the python package *prov*, developed by Huynh, is used. By having the capabilities of reading and writing external documents in PROV-JSON and PROV-XML serializations, it can fulfil this requirement. In addition, the package offers the possibility to create new provenance documents and records as objects according to the PROV standard. These objects can than be used to exchange provenance information in the prototype itself. In addition, it is possible to convert these objects into a graph representation, which can be used to filter the document as described in the algorithms of the concepts.

Using Python also yields the capabilities to create a prototype by using the concept of object oriented programming. By setting up a base class for the concepts, its is possible to fulfil the requirement P.3 of having a generalized API for all concepts. Subsequently, deriving the concepts from the base class leads to a code structure, which fulfils requirement P.4. In combination with internal unittest framework of Python, each concept can be tested independently, ensuring valid measurements on different subjects, such as performance.

5.2 Software Architecture

5.2.1 Packages

According to the definitions described above and with respect to the concept algorithms, a package structure for the prototype was specified. The core package consists of five subsequent packages, which are separated by their functions (Figure 5.1). The most important one is the *clients* package, which holds all classes representing the three concepts and the base class they are derived from. Since the accounts are acting differently on the blockchain in each concept, a second package is specified. It holds the base class and the derived classes, which respresent the more specialised accounts from the concepts To reuse already introduced accounts, a local database is needed to manage the attributed public/private keys and associated transaction ids. Since various databases are possible, the separate package *local_stores* was specified. Lastly, a *utils* and *exception* package was specified to outsource common functionalities and exceptions needed in each concept. Beside the core package structure, a second package for tests was defined which holds all unit tests for each class in the core package.

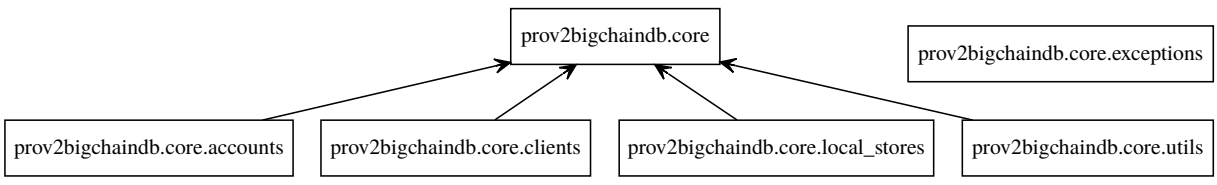


Figure 5.1: Package structure in prov2bigchaindb

5.2.2 Basic Classes

After setting up the overall package structure, all basic classes were defined in more detail (Figure 5.2). These classes are later used to derive the specialised classes for each concept.

SqliteStore First the class *SqliteStore* was introduced to provide a common interface to write and read account data from a single account table in a Sqlite database. By offering the methods `write_account()`, `get_account()` and `write_tx_id()` , and due to the prototype character of the application, the class was kept as simple as possible. Each object with access to an instance of this class can read or write new data to or from the table. Since the table contains the key pair and the transaction id of each account the class must be implemented in a more secure way, when used in a real application.

BaseAccount According to the algorithms, each concept requires one or more accounts to interact with the same kind of node. Therefore, the *BaseAccount* class was designed to contain all common attributes. These are a unique account id, a public/private key pair, an optional transaction id as well as a reference to the account database. Since all accounts need to create and transfer assets to a BigchainDB node, the two private methods `_create_asset()` and `_transfer_asset()` are placed in this class. The *BaseAccount* class also bears responsibility to create a new account in the database, if necessary. Due to different approaches of how accounts are handled by the concepts and which data is associated with an account, no further attributes or methods are defined in the base class

BaseClient In order to define public methods, which each derived client must re-implement to provide a consistent API, the *BaseClient* class was specified These are the abstract methods `save_document()` and `get_document()`. The method `test_transaction` was introduced to enable the clients to validate received transaction against BigchainDB. In addition, the class contains attributes for the account database and a connection pool. The decision to put both attributes into the base class was made for the reasons outline hereafter. Due to the concept design, the number of accounts with connections to a BigchainDB node cannot be estimated before parsing the provided provenance records. To avoid a large amount of parallel connections in worst cases, and to offer the possibility of easier parallelisation, the connection handling was centralized. This is also the case for connections with the local database.

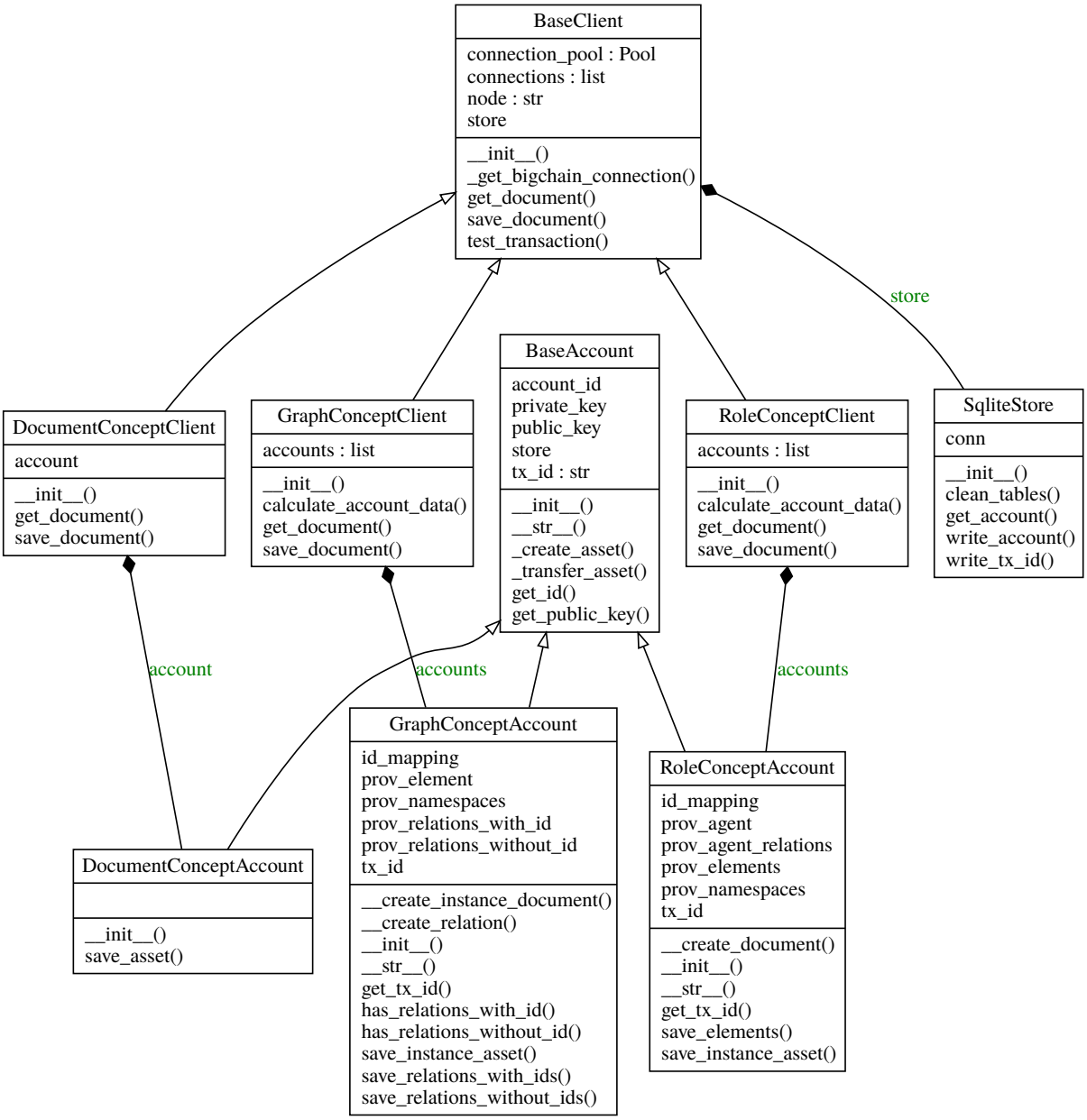


Figure 5.2: Class diagram for prov2bigchaindb

5.3 Implementation Details

5.3.1 Create and Transfer Assets

Creating and transferring assets between accounts via transactions is important in each concept. Therefore, this section will give an overview of how transactions are created and transferred using the python BigchainDB driver.

Create For creating an assets the method `_create_asset()`, defined in the BaseAccount class, was used throughout the prototypye. As depicted in line 1, the methods take three parameters (Listing 5.1). The first parameter is the connection to a BigchainDB node. The second and third parameters are python dictionaries containing the asset and optional metadata. By passing the public key, asset, metadata and the transaction type to the methods on line 5, a prepared transactions dictionary with the desired type is generated by the driver. Afterwards the transaction is fullfilled with the accounts private key. This step involves the generation of the transaction hash, which is used as id in the following process and subsequently the transaction is sent to a BigchchainDB node. The returned transaction can than be checked for any changes during the transmission to the node. Since the *send* methods are implemented as non-blocking, the returned transaction must not be successfully included in a block at this point. If the check on line 10 returns true, a dictionary with the transaction is returned.

```
1          def _create_asset(self, bdb_connection: BigchainDB, asset: dict, metadata: dict = None) ->
   ↪   dict:
2            if metadata is None:
3                    metadata = {}
4            prepared_creation_tx =
   ↪   bdb_connection.transactions.prepare(operation='CREATE',signers=self.public_key, asset=asset,
   ↪   metadata=metadata)
5
6            fulfilled_creation_tx = bdb_connection.transactions.fulfill(prepared_creation_tx,
   ↪   private_keys=self.private_key)
7
8            sent_creation_tx = bdb_connection.transactions.send(fulfilled_creation_tx)
9            if fulfilled_creation_tx != sent_creation_tx:
10                   raise exceptions.CreateRecordException()
11           return sent_creation_tx
```

Listing 5.1: Method _create_asset() function in BaseAccount class

As shown in the example transaction from line 2 to 10, the asset dictionary was placed under the key *asset* (Listing 5.2). In this case the key *prov* contains the serialized provenance about the PROV relation `wasAttributedTo`, as it gets written to BigchainDB in the graph-based concept. The transactions with provenance about the attributed types are referenced under the key *map* by their transaction ids. The *id* field contains the transaction id, which identifies the transaction. As expected *input* dictionary on line 12 contains the signature and the public of the account who created the transaction. Since the asset is owned by the same account in the first place, the *outputs* dictonary contains the same public key.

Transfer The BaseAccount class also implements the `_transfer_asset()` method, which allows the transfer of an asset to another account (Listing 5.3). As depicted on line 1, the methods takes four parameters. These are the connection to the node, the public key of the receiving account, the transactions to transfer and an optional dictionary containing metadata. First the asset is created and filled with the transaction id from the passed transaction (Line 5). Subsequently, the transaction input is configured by using the transaction output from the prior

```
1  {
2      "asset": {
3          "data": {
4              "map": {
5                  "ex:Ag2": "e25e59c734245c817fbf6da44a70cb30c1b86a0d9d09eb4ec587b10ec7810106",
6                  "ex:E2": "969c83e4b9377e97b771d28748b0250015af60d3e9539ab8676fac8b579f0b1b"
7              },
8              "prov": "{\"wasAttributedTo\": {\"_:id1\": {\"prov:entity\": \"ex:E2\", \"prov:agent\":
   ↪  \"ex:Ag2\"}}, \"prefix\": {\"pre_0\": \"https://www.w3.org/ns/prov#\", \"ex\":
   ↪  \"https://example.org/\"}}"
9          }
10     },
11     "id": "50cb5bbfb5a4abdfaa4e4a9f927d23c462650bcac94bd34fc0bed53d1934e9d3",
12     "inputs": [
13         {
14             "fulfillment": "cf:4:ZgxWoxuyXkM1nVcVdYfejJy_AN0FJ1OjpJ8dqoxiEHT52e-GOCLLZSM...",
15             "fulfills": null,
16             "owners_before": [
17                 "7sMVr3hXEtYrmThypr2MK5fHnZhGA366zXS1kXvrLu5M"
18             ]
19         }
20     ],
21     "metadata": {
22         "relation": "ex:E2->ex:Ag2"
23     },
24     "operation": "CREATE",
25     "outputs": [
26         {
27             "amount": 1,
28             "condition": {
29                 "details": {
30                     "bitmask": 32,
31                     "public_key": "7sMVr3hXEtYrmThypr2MK5fHnZhGA366zXS1kXvrLu5M",
32                     "signature": null,
33                     "type": "fulfillment",
34                     "type_id": 4
35                 },
36                 "uri": "cc:4:20:ZgxWoxuyXkM1nVcVdYfejJy_AN0FJ1OjpJ8dqoxiEHQ:96"
37             },
38             "public_keys": [
39                 "7sMVr3hXEtYrmThypr2MK5fHnZhGA366zXS1kXvrLu5M"
40             ]
41         }
42     ],
43     "version": "0.9"
44 }
```

Listing 5.2: Create transaction taken from a block in RethinkDB

transaction (Line 6-15). Afterwards the transaction should be assembled with help of the *prepare* function from the bigchaindb driver (Line 17-22). To create the desired transaction type the parameter *operation* is set to TRANSFER. The public key of the recipient is passed together with asset, metadata and prepared inputs. After signing the transaction with private key of the account, the transaction is send to a BigchainDB node (Line 24-26). If the transmission was successfull, the method returns a dictionary including the transaction.

After transferring the transaction to a BigchainDB node, the transaction is included into a block. The following listing shows the input section now contains the previous transaction id and the index of the output it refers to (Listing 5.4). The fulfilment or signature on line 8 changed in order to prove the ownership of the previous output. To transfer the ownership of the asset to the next owner, the public key is placed in the output section on line 28.

```
1           def _transfer_asset(self, bdb_connection: BigchainDB, recipient_pub_key: str, tx: dict,
    ↪   metadata: dict = None) -> dict:
2           if metadata is None:
3                   metadata = {}
4           transfer_asset = {'id': tx['id']}
5           output_index = 0
6           output = tx['outputs'][output_index]
7           transfer_input = {
8                   'fulfillment': output['condition']['details'],
9                   'fulfills': {
10                          'output': output_index,
11                          'txid': tx['id']
12                  },
13                  'owners_before': output['public_keys']
14          }
15
16          prepared_transfer_tx = bdb_connection.transactions.prepare(
17          operation='TRANSFER',
18          asset=transfer_asset,
19          metadata=metadata,
20          inputs=transfer_input,
21          recipients=recipient_pub_key)
22
23          fulfilled_transfer_tx = bdb_connection.transactions.fulfill(prepared_transfer_tx,
24
25
26          sent_transfer_tx = bdb_connection.transactions.send(fulfilled_transfer_tx)
27          if fulfilled_transfer_tx != sent_transfer_tx:
28                  raise exceptions.CreateRecordException()
29          return sent_transfer_tx
```

Listing 5.3: Method _transfer_asset() function in BaseAccount class

5.3.2 Document-based Concept

For implementing the document-based concept the classes DocumentConceptClient and DocumentConceptAccount were derived from the base classes. By instantiating a new DocumentConceptClient, a single DocumentConceptAccount object is created or loaded from the sqlilte database based on the given account id.

To store a PROV document the method `save_document()` is called. First the passed document gets converted into a ProvDocument object with help of the prov package, if needed. Afterwards the object is serialized to PROV-JSON and put into an asset dictionary by using its serialized method. This approach ensures valid representations of provenance data in every transaction and provides the ability to deserialize the information without prior calculations. Subsequently, the asset dictionary is provided to the account object by calling its `save_asset()` method. Since, no further processing is need the `_create_asset()` method from the base class is called. After the transaction id is returned, the function `wait_until_valid` from the utils package is called. The purpose of this method is to request the status of a transaction from BigChainDB until the status has changed to valid and the transaction is included into a block. The method throws an exception after a maximum of requests are reached, in order to prevent an infinite loop. Because of the following call to the `_transfer_asset()` method, it is necessary to wait for the previous transaction to get incorporated into a block. If this method is called with an invalid transaction, it will fail due to the consensus rules of BigchainDB. The transaction id return, by the successful transfer method, is then return to the DocumentConceptClient, which in turn returns the id.

In order to query a document from BigchainDB the method `get_document()` on the client object is called with a transaction id. Since queries on BigchainDB do not require an account, the id is immediately used to retrieve the corresponding transaction with the BigchainDB driver

```
{
    "asset": {
        "id": "50cb5bbfb5a4abdfaa4e4a9f927d23c462650bcac94bd34fc0bed53d1934e9d3"
    },
    "id": "12d5f4902ce83fcfc7e06d0e211201b480c586a5d1c34ad82053474fefd1bc31",
    "inputs": [
        {
            "fulfillment": "cf:4:ZgxWoxuyXkM1nVcVdYfejJy_AN0FJ1OjpJ8dqoxiEHTViJBrLQ2...",
            "fulfills": {
                "output": 0,
                "txid": "50cb5bbfb5a4abdfaa4e4a9f927d23c462650bcac94bd34fc0bed53d1934e9d3"
            },
            "owners_before": [
                "7sMVr3hXEtYrmThypr2MK5fHnZhGA366zXS1kXvrLu5M"
            ]
        }
    ],
    "metadata": {
        "relation": "ex:E2->ex:Ag2",
    },
    "operation": "TRANSFER",
    "outputs": [
        {
            "amount": 1,
            "condition": {
                "details": {
                    "bitmask": 32,
                    "public_key": "Heow8xCwCPEnedBGdBSiWyvyrnLYXac4FM5T3oAHyetc",
                    "signature": null,
                    "type": "fulfillment",
                    "type_id": 4
                },
                "uri": "cc:4:20:92qMa88eyT6MWR8yW3GIdEnyUHHIQ5vRj5cKYpw8JMU:96"
            },
            "public_keys": [
                "Heow8xCwCPEnedBGdBSiWyvyrnLYXac4FM5T3oAHyetc"
            ]
        }
    ],
    "version": "0.9"
}
```


Listing 5.4: Transfer transaction taken from a block in RethinkDB

method `retrieve()`. Because the returned transaction was not checked in the save method, it is checked against BigchainDB to prove the block and transaction as valid. The step ensures that only trustworthy provenance is returned regardless of the storing process. Due to the implementation of the `save_document()` method, the first transaction returned is of type TRANSFER and does not include the provenance data. Therefore, another request with the transaction id referenced in the previous asset is needed to retrieve the provenance. Afterwards the returned transaction is validated and the contained provenance is deserialized into a ProvDocument object and returned.

5.3.3 Graph-based Concept

Similar to the previous implementation, the graph-based concept derived the classes `GraphConceptClient` and `GraphConceptAccount` from their base classes. Since accounts are generated based upon the provided provenance document, an empty list is created when the GraphConceptClient object is instantiated.

In order to store new PROV statements the public method `save_document()` is called. First the provided document is transformed into a ProvDocument object. According to the algorithm

the object is than used to extract the PROV elements and all related relations by calling the static method `calculate_account_data()`. The method returns a list of triples, which contain the ProvElement object, a dictionary with ProvRelation objects and a list of namespaces. Afterwards, a dictionary containing a mapping between identifiers of relations and transaction ids is created from it. Subsequently, the list of triples is used to create an account object for each entry by passing the triple and the references to the mapping and the database object to the constructor of the GraphConceptAccount class. Within the initialization of the account object, the constructor of the base class is called with the identifier from the ProvElement object. This ensures that each account is represented by a unique PROV identifier. The account object is then put into a list and the method `save_instance_asset()` is called on the account object, creating a transaction holding the provenance about the ProvElement. A ProvDocument object including all PROV namespaces and the ProvElement is thus generated. Subsequently, the object is serialized to PROV-JSON and sent to the BigChainDB node. The asset is created and transferred, as in the previous concept. In contrast to the proposed algorithm, the transaction is not optimized in size due to the capabilities of BigchainDB After the successful inclusion of the transaction, its id is used to update the account entry in the local database and returned to the caller. The client finally appends the transaction id to a list, containing all id that together represent the document in the database. Following this step, the method `save_relations_with_ids()` is called on each account having relations with identifiers. For each relation the mapping used in the transaction and a ProvDocument object are generated. The mapping is assembled using the identifier mapping and the mapping provided by the local database. The mapping and serialized ProvDocument are put into the asset and send to the node using the usual methods from the base class hereafter. The returned transaction id is then set as value in the shared identifier mapping for later use. In addition, the transaction id is put into a list, which collects all ids generated in the loop. After all relations are created, the list is returned to the client object and appended to transaction list. The next step according to the algorithm is to create all relation without identifier. Therefore, the method `save_relations_without_ids()` is called to store all other relations with a similar approach as explained above. Lastly, all transaction ids are returned to caller using a list.

In order to retrieve a document from a BigchainDB node, a list of transaction ids must be passed to the method `get_document()` , defined in the GraphConceptClient class. This method works similarly to the method in DocumentConceptClient, but loops over each transaction id. All retrieved assets are deserialized to a temporary ProvDocument object, which in turn is used to collect all records into one ProvDocument. This Document is then returned to the calling method.

5.3.4 Role-based Concept

To implement the role-based concept the classes `RoleConceptClient` and `RoleConceptAccount` were derived from their base classes. As in the previous concept, an empty list for the accounts is created when an object of the RoleConceptClient class is instantiated. By calling its `save_document()` method, the given provenance statements are transformed into a ProvDocument object

As in the graph-based concept implementation, its static method `calculate_account_data()` is called to transform the ProvDocument into a data structure which can be used to create the needed accounts. In contrast to the method of the previous implementation, the method returns a list of quadruplet. Each quadruplet contains an ProvAgent object, a list of all its outgoing ProvRelations objects, a dictionary with all ProvElements objects associated or attributed to the Agent including all their outgoing ProvRelation objects, and a list of PROV namespaces. The method itself converts the ProcDocument object into a graph representation containing nodes

and edges. The nodes are than split up in two lists. The first list contains all ProvAgent objects where the second list contains all other ProvElement objects. Afterwards all the ProvElements are checked to see whether a relation to one of the ProvAgent's exists. If this is not the case for one of the ProvElements, the method will throw an exception since these ProvDocuments can't be stored according to the concept's design. Next, the method assembles the dictionary with the ProvElement object for each ProvAgent. This is done by directly using methods provided by networkx package, which is used by the prov package to represent the graph. Subsequently, the returned list of quadruplets is iterated to instantiate account objects using the RoleConceptAccount class. Thereafter, each account creates its initial transaction by getting called on the method. `save_instance_asset()`. This methods creates a ProvDocument object using the ProvAgent and its ProvRelations. The transaction is then issued to a BigchainDB node using the same technique as in the graph-based concept. The resulting id is returned to the client instance and included in a list of transactions. The final step is to create all other ProvElements calling the method `save_elements()`, which works similar to the previous method. The only difference is that more than one ProvElement might be processed in the method. Therefore, the method returns a list of transactions. After all ProvElements are successfully included into a BigchainDB node, the clients returns a list of transaction ids.

The current implementation lacks the features of a direct mapping between PROV identifiers and external transaction ids, as proposed in the concept section. It was ommited due to difficulties and complexity which come with the ordering of elements and relations prior to the creation of the transaction. Since all transaction ids are known, ids and all provenance is stored in its valid PROV-JSON representation, and querying of assets is still possible. the implementation of the method `get_document()` is equal to the method defined in the GraphConceptClient class.

5.3.5 System Level Unit Tests

To test all components of each concept together, three system level test cases were implemented beside others. These tests were used during the development process and for use later, in tests on performance. To illustrate how the system level test cases are designed, the test case of the graph-based concept is explained (Listing 5.5). By executing the test case, a specific ProvDocument object is created from a string containing PROV statements. In this case the document *simple2* is used, which corresponds to the file *test-example.json* from the asset directory. The file contains provenance that is similar to the examples used throughout the concept design chapter. Additional relations were added, in order to represent edge cases. Afterwards, a client object is initialised and the method `save_document()` is called with the ProvDocument. If the document is successfully saved, the `get_document()` method is called with the returned transaction ids. As explained, this method returns a ProvDocument object. To test the result the given object is compared with the returned object. In addition, the amount of records stored in the objects is compared.

```
1  def test_simple2_prov_doc(self):
2          prov_document = utils.to_prov_document(content=self.test_prov_files["simple2"])
3          graph_client = clients.GraphConceptClient(host=self.host, port=self.port)
4          tx_ids = graph_client.save_document(prov_document)
5          doc = graph_client.get_document(tx_ids)
6          self.assertEqual(len(prov_document.get_records()), len(doc.get_records()))
7          self.assertEqual(prov_document, doc)
```

Listing 5.5: System Level Unit Test for GraphConceptClient

Chapter 6

Test and Analysis

6.1 Environment

To execute the tests in an realistic and independent environment, a dedicated network of BigchainDB nodes was created. Since the decision on valid block in BigchainDB is based upon majority votes, a network of $2n + 1$ nodes must be guaranteed. Therefore, five BigchainDB and RethinkDB nodes were deployed on five virtual machines on a single VMWare host (Figure 6.1). Each machine was provided with two virtual Intel Xeon E5520 cores and 4GB RAM. As stated in the introduction under the section BigchainDB, Ubuntu 16.04 LTS x86_64 Release 2 was installed. The same operating system was used on a virtual machine running with eight Intel i7-3720QM cores and 8GB RAM. This machine was utilized as host for the software under test and connected to one of the nodes during a test run.

On each server, the RethinkDB nodes was first configured to interconnect with other nodes by using the RethinkDB's cluster protocol. Afterwards, each BigchainDB node was provided with a public and private key, and the keyring was filled with the public keys of all other nodes. (Listing 6.1). To have access to the RethinkDB cluster, each node was configured to connect with the local RethinkDB instance. In addition, the BigchainDB HTTP-API was made public on port 9984.

```
{
        "backlog_reassign_delay": 120,
        "database": {
                "backend": "rethinkdb",
                "host":   "localhost",
                "name":   "bigchain1",
                "port":   28015
        },
        "keypair": {
                "public":  "Bb9gCjgtz84uFhS9DXsGue2657WrraibfdeGE7BRRi2a",
                "private": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
        },
        "keyring": ["GkjPgiyWWhuewhExuubpt4rnMWMMPorQKsgF5aJihZku",
                                "4ihZoxhtT8kg2NZ3jMsYPgpFWnNbuy54GYwSeL1k6wQz",
                                "Bui1CMuNzFFosu1C5cCx7EuoLm9z6BhiNYDMBZ6ySWoe",
                                "HLSB55kmSJPiTEeRsz1n5QdnsiNuEtnU5dbcPsUMAUgg"],
        "server": {
        "workers": null,
        "threads": null,
        "bind":    "0.0.0.0:9984"
        }
}
```

Listing 6.1: BigchainDB Configuration File

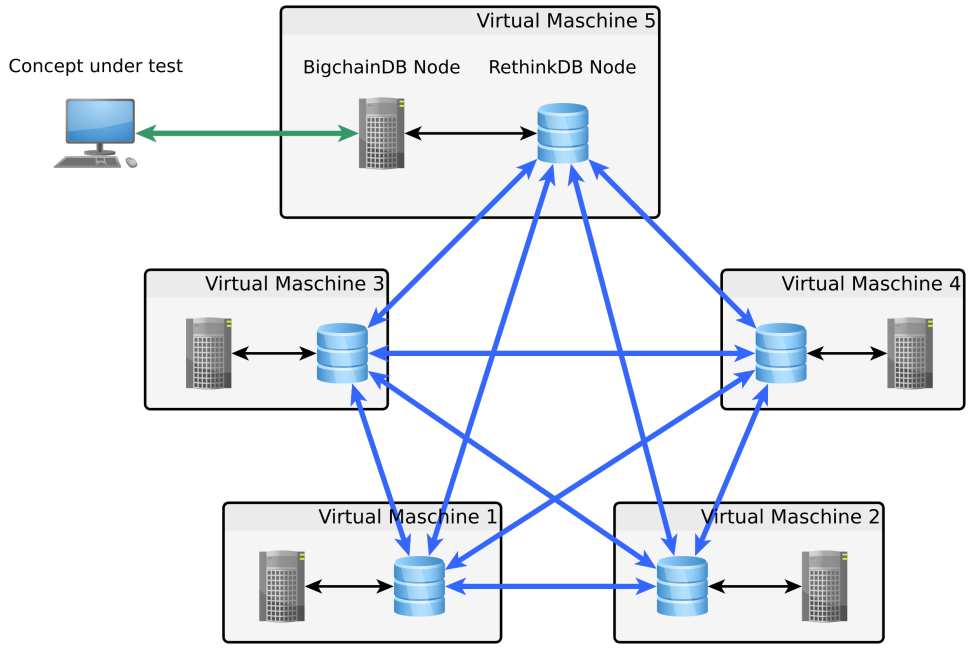


Figure 6.1: Test Environment with software under test

6.2 Test Procedure

Based upon the environment above, a test procedure was created to reduce side effects and ensure comparable test results. Prior to the test setup, all RethinkDB nodes were started once. The following procedure put forward was executed four times for each concept, with a system level test case.

Setup First, the database with all tables was created in RethinkDB by a BigchainDB task. Subsequently, the RethinkDB cluster was configured to split the database in five shards across all nodes. In addition, the replication factor was set to three, which ensured a working RethinkDB failover. After all RethinkDB nodes were synchronised their database, all BigchainDB nodes were started. After the BigchainDB nodes voted the block with the genesis transaction valid, the test itself was executed.

Execution To measure the performance and behaviour the concepts, the unit test framework was started with the desired test case from within the Pycharm IDE using the profiling option. This option activated a profiler from Pythons cProfile package to collects statistics during the runtime of the test, including information about function calls and measurements about their execution time. After completion of the test the collected measurements were saved to a *pstat* file for later analysis.

Tear Down To ensure the same starting point for each test run, all BigchainDB nodes were shutdown and the database in the RethinkDB cluster was deleted.

6.3 Results

The following section will present an analysis about the results gained from the collected measurement in the performance tests. First, all concepts will be discussed in detail to point out

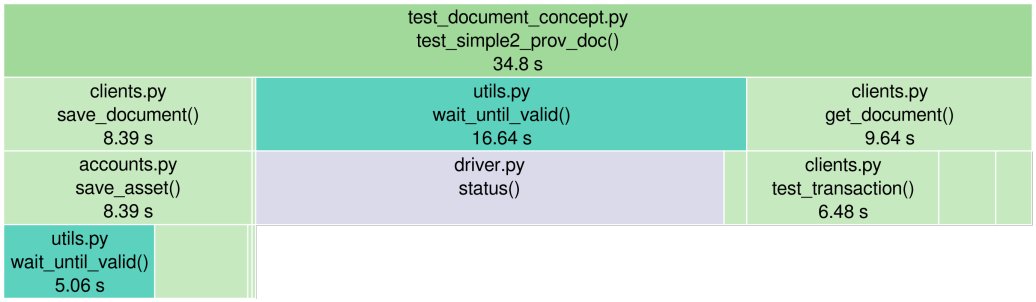


Figure 6.2: Testrun 1 – Document-based Concept

their behaviour and provide a comparison to some aspects made in the concept introduction. The discussion is followed by a general comparison of all concepts, covering the differences in performance.

6.3.1 Document-based Concept

All tests of the document based concept showed a measured runtime between 34.8 and 43.4 seconds and were able to save and retrieve the document in each run. As illustrated in the example of the first test run, with its accumulated 21.70 seconds, the function `wait_until_valid()` used up the majority of the total execution time (Figure 6.2). After sending the initial asset in the method `save_account()`, it took 5.06 seconds to incorporate the first transaction into the blockchain table. Consequently, transferring both transactions and creating the asset took about 3.3 seconds. But after sending the second transaction, the incorporation took about 16.64 seconds, which is significantly longer than expected. Since only one transaction was sent to the test environment beforehand and no other transaction was processed in between, it is assumed that the RethinkDB cluster may cause the delay. Variation in timing because of the `wait_until_valid()` method, was recognised in all tests of the concept. Being executed for approximately 10 seconds, the method `get_document()` was almost constant during all tests. The same is true for the method `save_document()`, which was also executed in approximately 10 seconds. Excluding the test on validity, the retrieving of both transactions needed about 3 seconds, which results in 1.5 seconds per transaction.

6.3.2 Graph-based Concept

During all tests the graph-based concept showed its capability to store and retrieve documents as intended. The measurements showed an overall runtime of the graph-based concept between 714 and 1290 seconds. This is the largest difference in runtime compared to all other concepts. On the one hand, the long runtime in general can be explained by the amount of transactions; on the other hand, this does not explain the huge difference in runtime between the tests. Due to the concept algorithm and used graph, the test stored 40 transactions into the blockchain table of RethinkDB. By having a total runtime of 546 seconds, the method `save_document()` in the fourth test does wait about 75 Percent of the time to receive a response from the BigchainDB node (Figure 6.3). As a result, each transaction waited more than 10 seconds to get included into the blockchain table. Therefore, the four seconds needed to create and transfer one asset stayed almost the same as in the document concept. Since, the measurements are limited to the accumulated time that was spent in a method or function, no detailed information is available about the time spent in methods called in loops. Consequently, it is not possible to detect if the time spent in the method `wait_until_valid()` is distributed equally or follows the same pattern as in the previous concept. The methods `save_instance_asset()` and

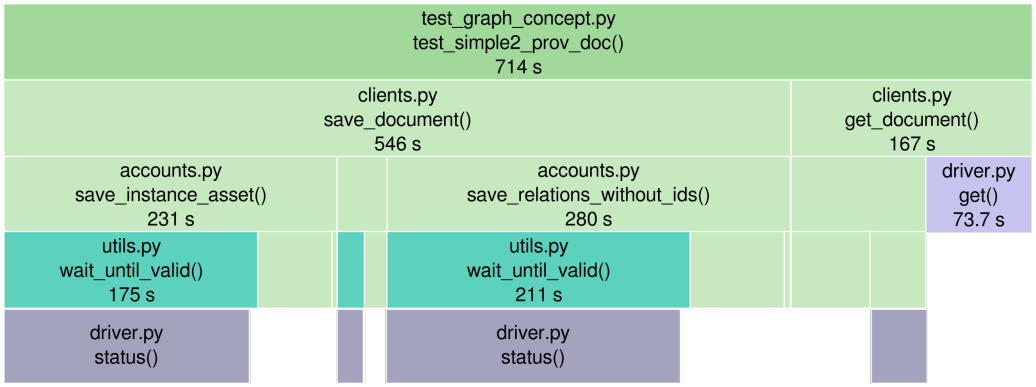


Figure 6.3: Testrun 4 – Graph-based concept

`save_relations_without_ids()` spend about 75 Percent of their execution time waiting until all transactions were processed. Since, the methods are called after another and the time to return from the `wait_until_valid()` method isn't significantly longer in the second method, its likely that the time to include a transaction in a block does not extend over the process. This assumption is underpinned by the measurements of the other test, which showed a similar pattern, but a significant difference in the overall time needed to store all transactions. In addition, the almost stable runtime between 154 and 206 seconds of the `get_document()` method in all tests showed that only the inclusion of assets seems to affect the time needed for the whole process. Since, the implementation of writing an asset to the blockchain table is identical in all concepts, the prior assumptions also point to problems with the RethinkDB cluster. But due to the significant difference of the time needed, it is possible that a process in BigchainDB may contribute to this problem.

6.3.3 Role-based Concept

Similar to the measurements of the previous concepts, the role-based concept proved its capability to store and retrieve documents to and from the blockchain table. With a runtime between 306 and 545 seconds due to only 16 transactions, the concept showed average results in performance, as expected. With approximately 3.4 seconds to create and transfer an asset, excluding the waiting process, the concept is comparable with those previously mentioned. This is also true for retrieving transactions from the blockchain. With an average of about two seconds, the measured time was close to the other concepts. As illustrated in the diagram of test 1, the concept showed similar behaviour regarding the `wait_until_valid()` method (Figure 6.4). The time consumed in waiting until all transactions were included into the blockchain table accumulated to about 70 Percent of the total time, on average over all tests. Therefore, the same assumption regarding the RethinkDB cluster must be made.

6.3.4 Problem Analysis

As already mentioned, the tests in all concepts were slowed down by waiting on transactions to get included into the blockchain. Therefore, the log files of the BigchainDB nodes were analysed during a test run with the following result. As soon as a new transaction arrived in the backlog table of the RethinkDB cluster, the transaction was picked by a random BigchainDB node and incorporated into a block. Subsequently, all others nodes were voting on the new block. Multiple reasons may lead to the rejection of a new block by a node. One out of these reasons is a signature error, which can occur if a node can't verify the signature of a block. In this case

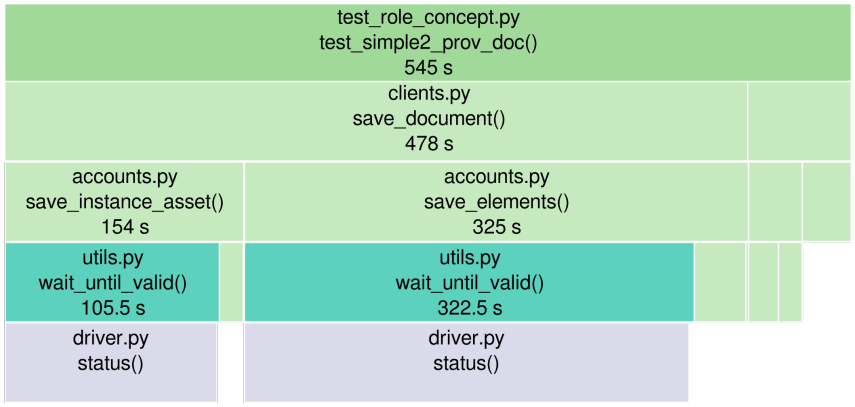


Figure 6.4: Testrun 1 – Role-based Concept

the transaction is put back into the backlog. The logs showed, that exactly this warning, about the error, occurred multiple times during the voting process on one block (Listing 6.2). The error itself can happen in two instances: First, a node does not know the public key of a node that created a new block; second, a node does not have access to the complete data set and therefore calculates the wrong hash values. Since all public keys were verified to be corrected on all nodes, only the second case might be plausible. Due to the shards and replica options set to the RethinkDB cluster, the data is distributed across all available nodes. Since only one node holds the primary shard, a piece of data must be requested by other nodes if needed. In some cases this might lead to a node that can't decide if a block is valid until the needed data is available. Another problem is that due to the design of software, exactly one block is created for each transaction. Additionally, the long voting process for one block might amplify the problem. Another possible but unlikely factor might be the strongly connected RethinkDB cluster, which can cause additional problems. For example, inefficient routing in specific network configurations can lead to additional delays. To prove this particular assumption, further research is required.

In conclusion, it must be stated that BigchainDB seems to cause the problem by initialising its voting process. Since BigchainDB is still in development, these problems may be solved in future versions. Therefore, the problem can be treated as side effect of the early software state.

```
1  WARNING:bigchaindb.consensus: Vote failed signature verification:
2  {"signature":
    ↪   "wb6o8Rh4rS2EuuuZYeGf6dnByH1QGutX6jNBgnEETkpeiwLxRmfqRCCyYFu2FTz2MT4TD3t44xcZKntLwYxEsPh",
3   "vote": {
4          "voting_for_block": "81710eb630a5c2313d028ef8057a5a7c3cd74464384b7ee4602cbff082ec6a84",
5          "invalid_reason": None,
6          "previous block": "20b006689dfc6516a70c5b47994a6fca4747b9b11324a56c1edfb3e5a769c957",
7          "is_block_valid": True,
8          "timestamp": "1490199519"},
9   "node pubkey":"6gaZbU9hMzHzxmSysHgNrG6gqG6wJRenM5ytCWkNXa8T"}
10 with voters:
11         ["Bb9gCjgtz84uFhS9DXsGue2657WrraibfdeGE7BRRi2a",
12          "GkjPgiyWWhuewhExuubpt4rnMWMMPorQKsgF5aJihZku",
13          "4ihZoxhtT8kg2NZ3jMsYPgpFWnNbuy54GYwSeL1k6wQz",
14          "HLSB55kmSJPiTEeRsz1n5QdnsiNuEtnU5dbcPsUMAUgg",
15          "Bui1CMuNzFFosu1C5cCx7EuoLm9z6BhiNYDMBZ6ySWoe"]
```

Listing 6.2: BigchainDB Configuration File

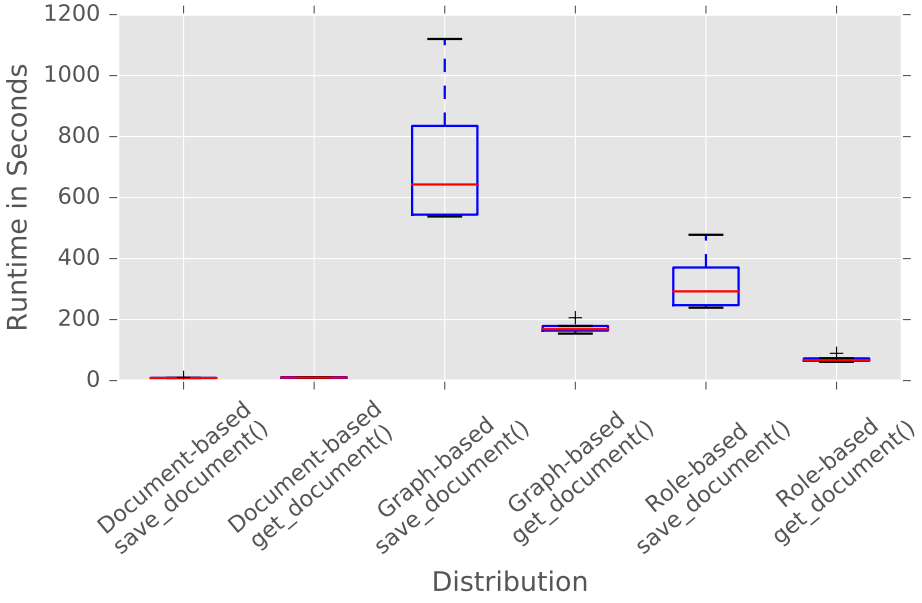


Figure 6.5: Measured performance of all tests combined split up by save and read methods

6.3.5 Summary

All concepts could prove their capability to store and retrieve provenance documents from the blockchain in general. As expected in the concept design, each concept showed different results in its performance. The measurement from all tests proved that the difference is strongly influenced by the amount of transactions transferred to the blockchain. This can be made clear if the run time of all three concept implementation is directly compared to each other (Figure 6.5). As depicted, the simple document based concept shows the most stable behaviour in writing and reading provenance. Due to the single transaction it is also the fastest way to store provenace in BigchainDB. By having a mean run time of about 280 seconds in storing provenance, the role-based concept showed an average result due to the fewer transactions required. The same is true for reading provenace from the blockchain. The graph-based concept showed the longest run time due to the amount of transactions needed. However, by using the same implementation for reading provenance from the blockchain, in the role-based implementation, a comparable performance was shown.

Chapter 7

Conclusion

Due to their variety of use cases and several concepts to prove the validity of blocks, blockchains offer different levels of resistance against tampering. As experienced by the author of this work, this was difficult to answer for blockchains implemented by private companies, which claim tamper-resistance or large file storage but does not provide any further documentation beside a high level with paper. Therefore, this work first analysed the different parameters which influence the tamper-resistance of open source currency-based blockchains and alternative blockchains. This allows for the selection of a suitable blockchain for an application with respect to the provided trustworthiness, security and performance. The current project proposed a solution for a wide variety of blockchains, utilizing three generalized concepts for storing provenance in its PROV representation into a blockchain. Consequently, the concepts are usable for the development of various applications based on the blockchain technology with focus on provenance. Since provenance deals with huge data sets and is mostly stored in private environments, the author of this work utilized the blockchain-like BigchainDB as a software prototype. Due to the flexibility of BigchainDB it was possible to implement each concept and prove its capability, advantages and disadvantages. In addition, use cases for each concepts were provided to the reader.

During the process of writing this thesis further possibilities for future research were discovered. The problems experienced by using RethinkDB in combination with BigchainDB suggests an implementation with the optional MongoDB should be considered in order to prove the measurements of the thesis. Further, both databases should be explicitly tested against tampering by altering the blockchains using the capabilities of the database. This would allow a more precise evaluation of the tamper-resistance BigchainDB must provide in order to offer trustworthy storage of provenance data. Since the role-based concept is limited to a subset of the PROV standard, further research is required to solve this problem. This is also true for the unsolved problem of finding a deterministic ordering, which is needed for a valid mapping between transactions. Regrading the current prototype, the database to store account data must be considered unsafe. Therefore, a secure solution must be implemented to prevent stealing of private keys.

Another possible research topic that arose is the implementation of the concepts using Ethereum. From the authors point of view, the graph-based concept should be used in this approach. By moving the program logic into contracts, additional security and control mechanisms as well as new types of automated queries are possible. Of special interest, is an analysis on the tamper-resistance influenced by Ethereum's turing-complete programming language.

References

[1] German Aerospace Center (DLR). *DLR at a glance*. Online. Retrieved: 23.11.2016. URL: www.dlr.de/dlr/en/desktopdefault.aspx/tabid-10443/.

[2] German Aerospace Center DLR. *Simulation and Software Technology*. Online. Retrieved: 23.11.2016. URL: http://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1177/.

[3] Michael Meinel. *BACARDI: Ein Katalog für Raumfahrtrückstände*. Online. Retrieved: 23.11.2016. URL: http://elib.dlr.de/86145/.

[4] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Online. Retrieved: 29.08.2016. 10/2008. URL: http://nakamotoinstitute.org/bitcoin/.

[5] Xiaoyun Wang and Hongbo Yu. "How to Break MD5 and Other Hash Functions". In: *Advances in Cryptology – EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings*. Ed. by Ronald Cramer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 19–35. ISBN: 978-3-540-32055-5. DOI: 10.1007/11426639_2. URL: http://dx.doi.org/10.1007/11426639_2.

[6] Hongbo Yu and Dongxia Bai. "Boomerang Attack on Step-Reduced SHA-512". In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 945.

[7] Maria Eichlseder, Florian Mendel, and Martin Schläffer. "Branching Heuristics in Differential Collision Search with Applications to SHA-512". In: *Fast Software Encryption: 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*. Ed. by Carlos Cid and Christian Rechberger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 473–488. ISBN: 978-3-662-46706-0. DOI: 10.1007/978-3-662-46706-0_24. URL: http://dx.doi.org/10.1007/978-3-662-46706-0_24.

[8] Ralph C. Merkle. "Secrey, Authentication, and Public Key Systems". Retrieved: 25.11.2016. dissertation. Stanford Univiersity, 1979. URL: http://www.merkle.com/papers/Thesis1979.pdf.

[9] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: http://doi.acm.org/10.1145/357172.357176.

[10] Paul Groth and Luc Moreau. *PROV-Overview*. Online. Retrieved: 29.08.2016. 2013. URL: https://www.w3.org/TR/prov-overview/.

[11] Luc Moreau and Paolo Missier et al. *PROV-DM: The PROV Data Model*. Online. Retrieved: 29.08.2016. 2013. URL: https://www.w3.org/TR/prov-dm/.

[12] Oxford Dictionaries. *Definition of provenance in English*. Online. Retrieved: 06.03.2017. URL: https://en.oxforddictionaries.com/definition/provenance.

[13] Wikipedia. *Provenance*. Online. Retrieved: 09.11.2016. 2016. URL: https://en.wikipedia.org/wiki/Provenance.

[14] Luc Moreau. "The Foundations for Provenance on the Web". In: *Foundations and Trends® in Web Science* 2.2–3 (2010), pp. 99–241. ISSN: 1555-077X. DOI: 10.1561/1800000010. URL: http://dx.doi.org/10.1561/1800000010.

[15] Ygil. *What Is Provenance*. Online. Retrieved: 09.11.2016. 2005. URL: https://www.w3.org/2005/Incubator/prov/wiki/What_Is_Provenance.

[16] openprovenance.org. *The OPM Provenance Model (OPM)*. Online. Retrieved: 15.11.2016. URL: http://openprovenance.org.

[17] Luc Moreau et al. "The Open Provenance Model core specification (v1.1)". Retrieved: 11.11.2016. June 2011. URL: http://eprints.soton.ac.uk/271449/.

[18] Luc Moreau et al. "The Rationale of PROV". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 35.4 (2015). ISSN: 1570-8268. URL: http://www.websemanticsjournal.org/index.php/ps/article/view/454.

[19] W3C. *Linking Across Provenance Bundles*. Online. Retrieved: 28.11.2016. 2013. URL: https://www.w3.org/TR/prov-sem/.

[20] Yolanda Gil and Simon Miles et al. *PROV Model Primer*. Online. Retrieved: 29.08.2016. 2013. URL: https://www.w3.org/TR/prov-primer/.

[21] Timothy Lebo and Satya Sahoo et al. *PROV-O: The PROV Ontology*. Online. Retrieved: 29.08.2016. 2013. URL: https://www.w3.org/TR/prov-o/.

[22] Luc Moreau. *PROV-XML: The PROV XML Schema*. Online. Retrieved: 22.11.2016. 2013. URL: https://www.w3.org/TR/prov-xml/.

[23] Trung Dong Huynh and Michael O. Jewell et al. *The PROV-JSON Serialization*. Online. Retrieved: 22.11.2016. 2013. URL: https://www.w3.org/Submission/prov-json/.

[24] Luc Moreau and Paolo Missier et al. *PROV-N: The Provenance Notation*. Online. Retrieved: 29.08.2016. 2013. URL: https://www.w3.org/TR/prov-n/.

[25] James Cheney and Paolo Missier et al. *Constraints of the PROV Data Model*. Online. Retrieved: 29.08.2016. 2013. URL: https://www.w3.org/TR/prov-constraints/.

[26] M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987 (Proposed Standard). Internet Engineering Task Force, Jan. 2005. URL: http://www.ietf.org/rfc/rfc3987.txt.

[27] Richard Cyganiak and DERI et al. *RDF Concepts and Abstract Syntax*. Online. Retrieved: 22.11.2016. 2014. URL: https://www.w3.org/TR/rdf-concepts/.

[28] Ragib Hasan and Joseph Tucek et al. *The Techniques and Challenges of Immutable Storage with Applications in Multimedia*. Online. Retrieved: 25.11.2016. URL: https://pdfs.semanticscholar.org/578f/f4957d4fa2e550ec2a819b6500820d7286cd.pdf.

[29] Wikipedia. *Immutable characteristic*. Retrieved: 25.11.2016. 2016. URL: https://en.wikipedia.org/w/index.php?title=Immutable_characteristic&oldid=723035023.

[30] Elizabeth Haubert et al. "Tamper-Resistant Storage Techniques for Multimedia Systems". In: *IN IS and T/SPIE INTERNATIONAL SYMPOSIUM ELECTRONIC IMAGING / STORAGE AND RETRIEVAL METHODS AND APPLICATIONS FOR MULTIMEDIA*. 2005, pp. 30–40.

[31] Wikipedia. *Blockchain (database)*. Online. Retrieved: 25.11.2016. 2016. URL: https://en.wikipedia.org/wiki/Blockchain_(database).

[32] Bitcoin.org. *Bitcoin Developer Guide*. Online. Retrieved: 17.01.2017. 2017. URL: https://bitcoin.org/en/developer-guide.

[33] Bitcoin. *Bitcoin Blockchain Releases*. Online. Retrieved: 17.01.2017. 2017. URL: https://github.com/bitcoin/bitcoin/releases.

[34] Blockchain.info. *Blockchain Size*. Online. Retrieved: 17.01.2017. 2017. URL: https://blockchain.info.

[35] 21.co. *Global Bitcoin Nodes Distribution*. Online. Retrieved: 17.01.2017. 2017. URL: https://bitnodes.21.co/.

[36] BitcoinWiki. *Network*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Network.

[37] A.M. Antonopoulos. *Mastering Bitcoin*. O'Reilly Media, Incorporated, 2014. ISBN: 978-1-449-37404-4. URL: http://chimera.labs.oreilly.com/books/1234000001802.

[38] BitcoinWiki. *Technical background of version 1 Bitcoin addresses*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses.

[39] BitcoinWiki. *Base58Check encoding*. Online. Retrieved: 17.01.2017. URL: https://en.bitcoin.it/wiki/Base58Check_encoding.

[40] BitcoinWiki. *Wallet*. Online. Retrieved: 07.03.2017. URL: https://en.bitcoin.it/wiki/Wallet.

[41] BitcoinWiki. *Transaction*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Transaction.

[42] BitcoinWiki. *Protocol rules*. Online. Retrieved: 26.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Protocol_rules.

[43] BitcoinWiki. *Script*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Script.

[44] BitcoinWiki. *Double-Spending*. Online. Retrieved: 07.03.2017. URL: https://en.bitcoin.it/wiki/Double-spending.

[45] BitcoinWiki. *Nonce*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Nonce.

[46] Bitcoin.org. *Bitcoin Developer Reference*. Online. Retrieved: 17.01.2017. 2017. URL: https://bitcoin.org/en/developer-reference.

[47] BitcoinWiki. *Proof of Work*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Proof_of_work.

[48] BitcoinWiki. *CVE-2010-5139 - Combined output overflow*. Online. Retrieved: 21.02.2017. Aug. 2010. URL: https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures#CVE-2010-5139.

[49] BitcoinWiki. *Majority Attack*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Majority_attack.

[50] BitcoinWiki. *Confirmation*. Online. Retrieved: 07.03.2017. URL: https://en.bitcoin.it/wiki/Confirmation.

[51] Wikipedia. *Litecoin*. Online. Retrieved: 07.03.2017. 2017. URL: https://en.wikipedia.org/wiki/Litecoin.

[52] Wikipedia. *Dodgecoin*. Online. Retrieved: 07.03.2017. 2017. URL: https://en.wikipedia.org/wiki/Dogecoin.

[53] Wikipedia. *Scrypt*. Online. Retrieved: 03.02.2017. 2017. URL: https://en.wikipedia.org/w/index.php?title=Scrypt&oldid=759490083.

[54] BitcoinWiki. *Scrypt proof of work*. Online. Retrieved: 03.02.2017. 2017. URL: https://en.bitcoin.it/wiki/Scrypt_proof_of_work.

[55] Wikipedia. *Proof-of-stake*. Online. Retrieved: 07.03.2017. 2017. URL: https://en.wikipedia.org/wiki/Proof-of-stake.

[56] Wikipedia. *Peercoin*. Online. Retrieved: 07.03.2017. 2017. URL: https://en.wikipedia.org/wiki/Peercoin.

[57] Sunny Kind and Scott Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. Online. Retrieved: 07.03.2017. Aug. 2012. URL: https://peercoin.net/assets/paper/peercoin-paper.pdf.

[58] Nikolai Hampton. "Understanding the blockchain hype: Why much of it is nothing more than snake oil and spin". In: *Computerworld* (Sept. 2016). Retrieved: 21.02.2017. URL: http://www.computerworld.com.au/article/606253/understanding-blockchain-hype-why-much-it-nothing-more-than-snake-oil-spin/.

[59] Bitcoin.org. *Bitcoin Core version 0.11.0 released*. Online. Retrieved: 21.02.2017. July 2015. URL: https://bitcoin.org/en/release/v0.11.0.

[60] BitcoinWiki. *Coinbase*. Online. Retrieved: 17.01.2017. 2017. URL: https://en.bitcoin.it/wiki/Coinbase.

[61] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. Online. Retrieved: 09.11.2016. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[62] Wikipedia. *Ethereum*. Online. Retrieved: 21.02.2017. 2017. URL: https://de.wikipedia.org/w/index.php?title=Ethereum&oldid=162227850.

[63] Ethereum Foundation. *The Homestead Release*. Online. Retrieved: 21.02.2017. URL: http://ethdocs.org/en/latest/.

[64] ETHDEV. *Ethereum Frontier Guide*. Online. Retrieved: 21.02.2017. URL: https://ethereum.gitbooks.io/frontier-guide/content/index.html.

[65] Ethereum Foundation. *Design Rationale*. Online. Retrieved: 21.02.2017. URL: https://github.com/ethereum/wiki/wiki/Design-Rationale.

[66] Ethereum Foundation. *Setting up private network or local cluster*. Online. Retrieved: 21.02.2017. URL: https://github.com/ethereum/go-ethereum/wiki/Setting-up-private-network-or-local-cluster.

[67] Gavin Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. Online. Retrieved: 21.02.2017. June 2014. URL: http://gavwood.com/paper.pdf.

[68] Morgen Peck. ""Hard Fork" Coming to Restore Ethereum Funds to Investors of Hacked DAO". In: *IEEE Spectrum* (July 2016). Retrieved: 212.02.2017. URL: http://spectrum.ieee.org/tech-talk/computing/networks/hacked-blockchain-fund-the-dao-chooses-a-hard-fork-to-redistribute-funds.

[69] *Solidity 0.4.10 documentation*. Online. Retrieved: 21.02.2017. URL: https://solidity.readthedocs.io/en/develop/.

[70] Ethereumbuilders. *Solidity Tutorials*. Online. Retrieved: 21.02.2017. URL: https://ethereumbuilders.gitbooks.io/guide/content/en/solidity_tutorials.html.

[71] Elena Dimitrova. *Writing robust smart contracts in Solidity*. Online. Retrieved: 21.02.2017. URL: https://blog.colony.io/writing-more-robust-smart-contracts-99ad0a11e948#.ovyyera31.

[72] *Ethereum Network Status*. Online. Retrieved: 18.02.2017. URL: https://ethstats.net/.

[73] ConsenSys. *Technical Introduction to Events and Logs in Ethereum*. Online. Retrieved: 21.02.2017. URL: https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e#.i2w4ivny7.

[74] Trent McConaghy and Rodolphe Marques et al. *BigchainDB: A Scalable Blockchain Database*. Online. Retrieved: 29.08.2016. 2016. URL: https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf.

[75] Ascribe. *About BigchainDB*. Online. Retrieved: 21.02.2017. URL: https://www.bigchaindb.com/about/.

[76] Ascribe. *BigchainDB Documentation*. Online. Retrieved: 21.02.2017. URL: https://docs.bigchaindb.com/en/latest/.

[77] OpenAssets. *Open Assets Protocol*. Online. Retrieved: 21.02.2017. URL: https://github.com/OpenAssets/open-assets-protocol.

[78] Colored-Coins. *The Colored Coins Protocol*. Online. Retrieved: 21.02.2017. URL: https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification.

[79] Trent McConaghy and David Holtzman. *Towards An Ownership Layer for the Internet*. Online. Retrieved: 21.02.2017. June 2015. URL: https://d1qjsxua1o9x03.cloudfront.net/live/trent@ascribe.io/ascribe%20whitepaper%2020150624/digitalwork/ascribe%20whitepaper%2020150624.pdf.

[80] G Irving and J Holden. "How blockchain-timestamped protocols could improve the trustworthiness of medical science [version 2; referees: 3 approved]". In: *F1000Research* 5.222 (2016). DOI: 10.12688/f1000research.8114.2.

[81] T Nugent, D Upton, and M Cimpoesu. "Improving data transparency in clinical trials using blockchain smart contracts [version 1; referees: 1 approved]". In: *F1000Research* 5.2541 (2016). DOI: 10.12688/f1000research.9756.1.

[82] Trung Dong Huynh and Luc Moreau. "ProvStore: a public provenance repository". June 2014. URL: http://eprints.soton.ac.uk/365509/.

List of Abbreviations

OPM	OPM Provenance Model
IPAW	International Provenance and Annotation Workshop
RDF	Ressource Description Framework
OWL2	Web Ontology Language in Version 2
UML	Unified Modeling Language
OOP	Object-oriented programming
IRI	Internationalized Resource Identifiers
DLR	German Aerospace Center
GSOC	German Space Operations Center
BACARDI	Backend Catalog for Relational Debris Information
SPV	Simplified Payment Verification
ECDSA	Elliptic Curve Digital Signature Algorithm
UTXO	Unspended Transaction Output
PROV-DM	PROV Data Model
PROV-O	PROV Ontology
PROV-N	PROV Notation
PROV-XML	PROV Extensible Markup Language
PROV-JSON	PROV JavaScript Object Notation
ASIC	Application Specific Integrated Circuit
PoS	Proof-of-Stake
PoW	Proof-of-Work
PoS	Proof-of-Stake
DAO	Decentralized Autonomous Organization
GPU	Graphics Processing Unit
EVM	Ethereum Virtual Machine
JSON	JavaScript Object Notation
HTTP	HyperText Transfer Protocol
IPFS	InterPlanetary File System
ReST	Representational State Transfer
API	Application Programming Interface
W3C	World Wide Web Consortium
XML	Extensible Markup Language

List of Figures

List of Tables

List of Listings

Appendix A

PROV Data Model

Table A.1: Overview PROV Components [11]

Type or Relation	Representation in the PROV-N notation	Component
Entity	(id, [attr1=val1, ...])	Component 1: Entities, Activities
Activity	(id, st, et, [attr1=val1, ...])	
Generation	wasGeneratedBy(id;e,a,t,attrs)	
Usage	used(id;a,e,t,attrs)	
Communication	wasInformedBy(id;a2,a1,attrs)	
Start	wasStartedBy(id;a2,e,a1,t,attrs)	
End	wasEndedBy(id;a2,e,a1,t,attrs)	
Invalidation	wasInvalidatedBy(id;e,a,t,attrs)	
Derivation	wasDerivedFrom(id; e2, e1, a, g2, u1, attrs)	Component 2: Derivations
Revision	... prov:type='prov:Revision' ...	
Quotation	... prov:type='prov:Quotation' ...	
Primary Source	... prov:type='prov:PrimarySource' ...	
Agent	(id, [attr1=val1, ...])	Component 3: Agents, Responsibility, Influence
Attribution	wasAttributedTo(id;e,ag,attr)	
Association	wasAssociatedWith(id;a,ag,pl,attrs)	
Delegation	actedOnBehalfOf(id;ag2,ag1,a,attrs)	
Plan	... prov:type='prov:Plan' ...	
Person	... prov:type='prov:Person' ...	
Organization	... prov:type='prov:Organization' ...	
SoftwareAgent	... prov:type='prov:SoftwareAgent' ...	
Influence	wasInfluencedBy(id;e2,e1,attrs)	
Bundle constructor	bundle id description_1 ... description_n endBundle	Component 4: Bundles
Bundle type	... prov:type='prov:Bundle' ...	
Alternate	alternateOf(alt1, alt2)	Component 5: Alternate
Specialization	specializationOf(infra, supra)	
Collection	... prov:type='prov:Collection' ...	Component 6: Collections
EmptyCollection	... prov:type='prov:EmptyCollection' ...	
Membership	hadMember(c,e)	

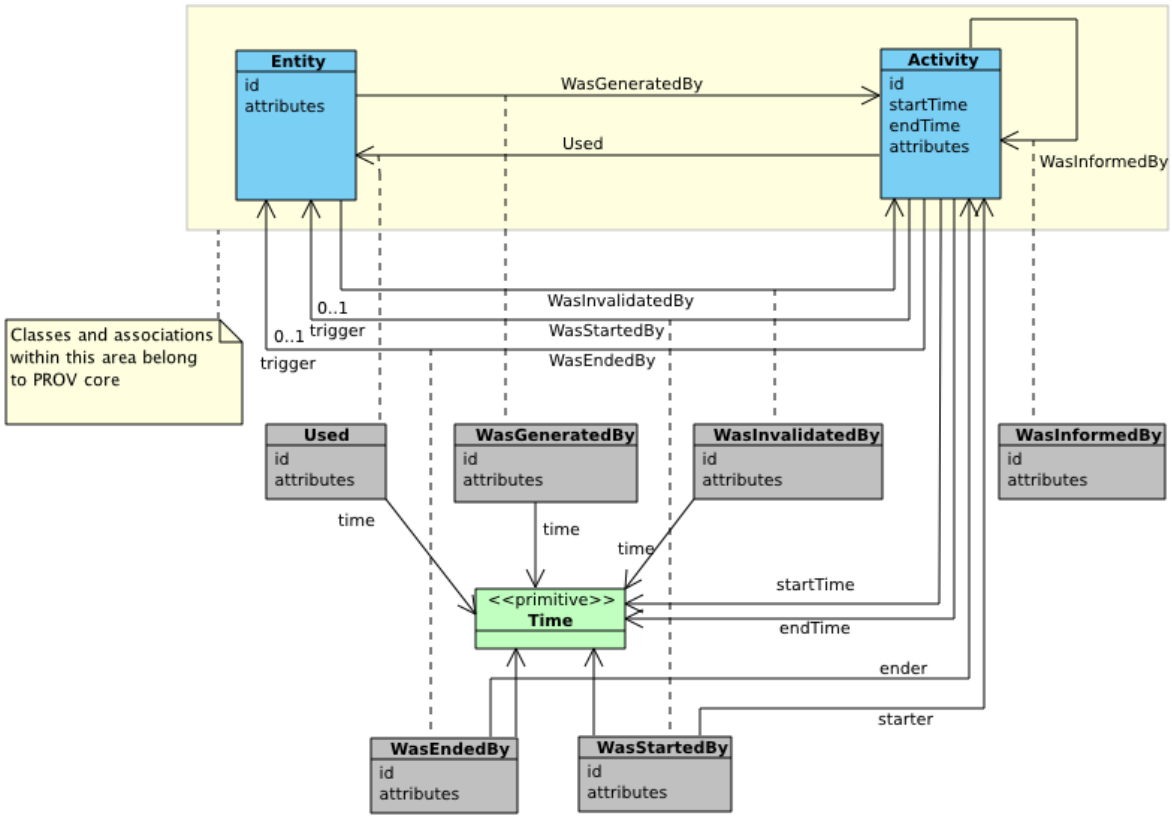


Figure A.1: UML Representation of Component 1 [11]

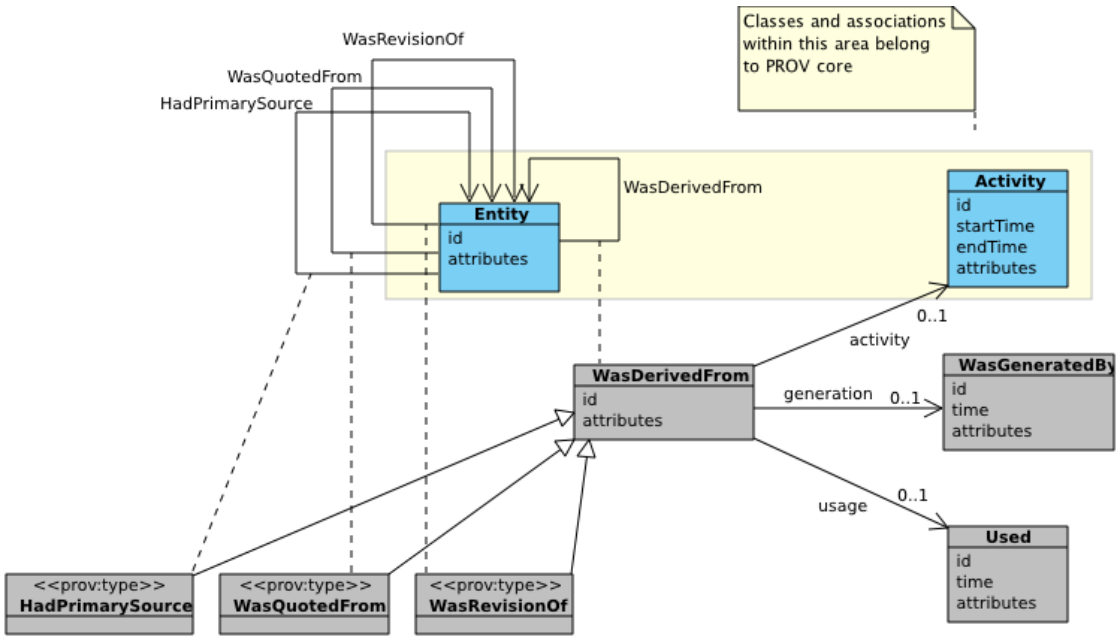


Figure A.2: UML Representation of Component 2 [11]

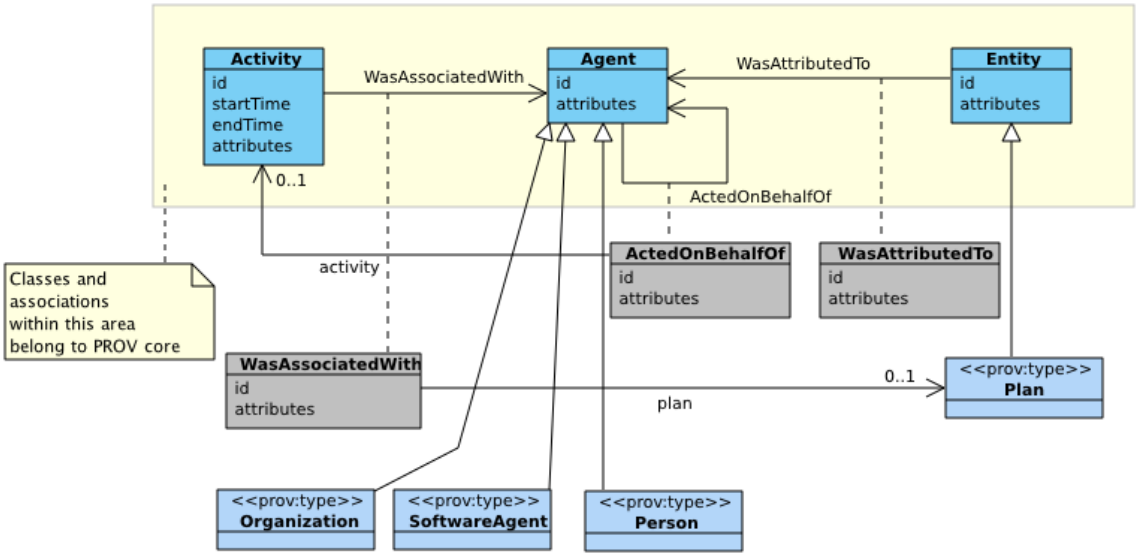


Figure A.3: UML Representation of Component 3a [11]

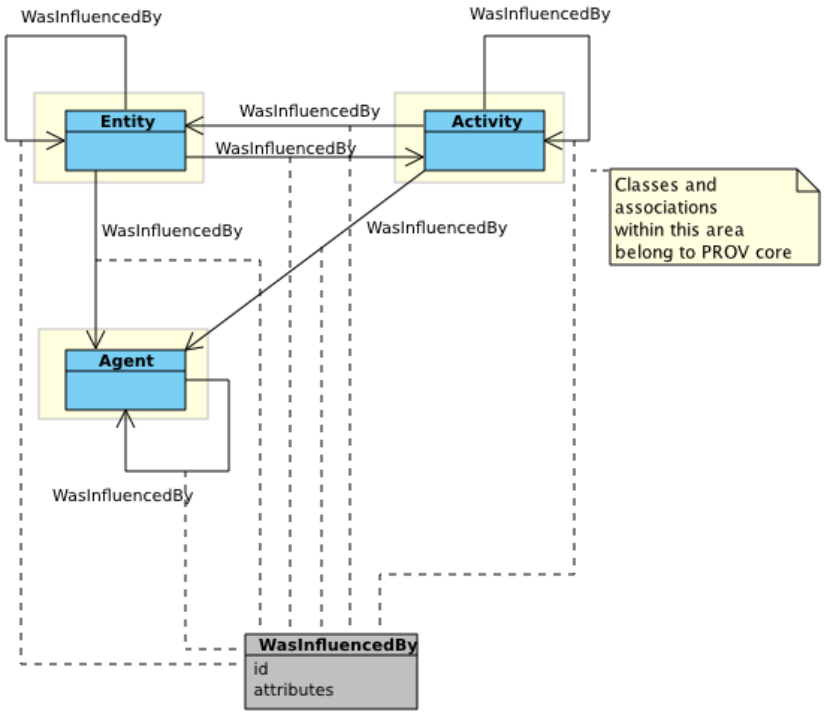


Figure A.4: UML Representation of Component 3b [11]

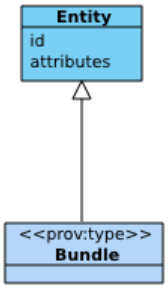


Figure A.5: UML Representation of Component 4 [11]

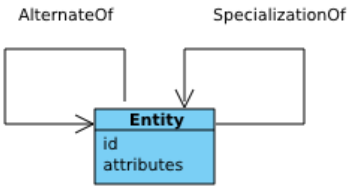


Figure A.6: UML Representation of Component 5 [11]

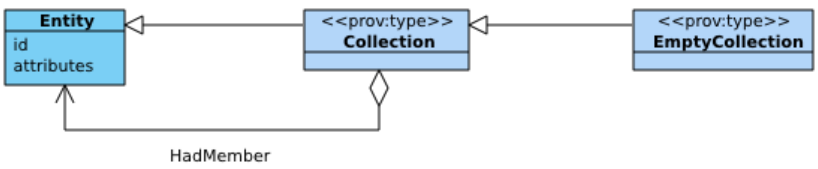


Figure A.7: UML Representation of Component 6 [11]

```
1  document
2     // Namespaces
3     default <https://example.org/0/>
4     prefix ex <https://example.org/1/>
5     prefix prov <https://www.w3.org/ns/prov#>
6     prefix ul <https://uni-leipzig.de/prov/>
7     prefix dlr <https://dlr.de/prov/>
8     prefix tr <https://www.w3.org/TR/2013/>
9
10    // Component 4
11    entity(ul:master-thesis-stoffers, [ prov:type='prov:Bundle' ])
12    bundle ul:master-thesis-stoffers
13
14       // Types
15       entity(ul:thesis-stoffers-finished, [ prov:type="ul:
            masterthesis"])
16       entity(ul:thesis-stoffers-20170316, [ ul:title="Trustworthy
             Provenance..." ])
17       entity(ul:thesis-stoffers-20170310)
18       entity(ul:milestone-1, [ prov:type="Plan" ])
19       entity(tr:REC-prov-dm-20130430, [ prov:label="The PROV Data
             Model" ])
20
21       activity(ul:worked-on-thesis, 2017-10-03T09:00:00,
            2016-10-03T09:30:00, [ prov:type='ul:edit' ])
22       activity(ex:print-thesis, -, -, [ prov:type='ex:print' ])
23
24       agent(ul:martin-stoffers, [ul:regno="3748896", ul:name="
            Martin Stoffers", prov:type='prov:Person' ])
25       agent(ul:michael-martin, [ul:name="Michael Martin", prov:
            type='prov:Person' ])
26       agent(ul:university-leipzig, [prov:type='prov:Organisation'
             ])
27       agent(dlr:andreas-schreiber, [ul:name="Andreas Schreiber",
            prov:type='prov:Person' ])
28       agent(dlr:dlr, [prov:type='prov:Organisation' ])
29       agent(ex:print-shop, [prov:type='prov:Organisation' ])
30
31       // Relations
32       // Component 1
33       wasGeneratedBy(ul:thesis-stoffers-20170316, ul:worked-on-
            thesis, -)
34       wasGeneratedBy(ul:thesis-stoffers-finished, ex:print-thesis
            , -)
35       used(tr:REC-prov-dm-20130430, ul:worked-on-thesis,
            2017-03-16T09:10:00)
36       used(ul:thesis-stoffers-20170316, ex:print-thesis, -)
37       wasInformedBy(ex:print-thesis, ul:worked-on-thesis)
38       wasStartedBy(ex:print-thesis, ul:thesis-stoffers-20170316,
            ul:worked-on-thesis, 2017-03-17T10:00:00)
39       wasEndedBy(ex:print-thesis, ul:thesis-stoffers-20170316, ul
            :worked-on-thesis, 2017-03-17T10:10:00)
```

Listing A.1: Full PROV-N Example (Part a)

```
1        // Component 2
2        wasInvalidatedBy(ul:thesis-stoffers-20170316, ex:print-
             thesis, -)
3        wasDerivedFrom(ul:thesis-stoffers-20170316, ul:thesis-
             stoffers-20170310, -, -, -, [ prov:type="prov:Revision"
              ])
4
5        // Component 3
6        wasAttributedTo(ul:thesis-stoffers-20170316, ul:martin-
             stoffers)
7        wasAttributedTo(ul:master-thesis-stoffers, ul:university-
             leipzig, -)
8        wasAssociatedWith(ul:worked-on-thesis, ul:martin-stoffers,
             ul:milestone-1)
9        wasAssociatedWith(ex:print-thesis, ex:print-shop, -)
10       actedOnBehalfOf(ex:print-shop, ul:martin-stoffers, ex:print
             -thesis)
11       actedOnBehalfOf(ul:michael-martin, ul:university-leipzig)
12       wasInfluencedBy(ul:master-thesis-stoffers, ul:michael-
             martin, [prov:role="ul:Supervisior"])
13       actedOnBehalfOf(dlr:andreas-schreiber, dlr:dlr)
14       wasInfluencedBy(ul:master-thesis-stoffers, dlr:andreas-
             schreiber, [prov:role="ul:Supervisior"])
15
16       // Component 5
17       specializationOf(ul:thesis-stoffers-20170316, ul:thesis-
             stoffers-finished)
18       specializationOf(ul:thesis-stoffers-20170310, ul:thesis-
             stoffers-finished)
19       alternateOf(ul:thesis-stoffers-20170316, ul:thesis-stoffers
             -20170310)
20
21       // Component 6
22       entity(ul:thesis-stoffers-versions, [prov:type='prov:
             Collection'])
23       hadMember(ul:thesis-stoffers-versions, ul:thesis-stoffers
             -20170316)
24       hadMember(ul:thesis-stoffers-versions, ul:thesis-stoffers
             -20170310)
25   endBundle
26 endDocument
```

Listing A.2: Full PROV-N Example (Part b)

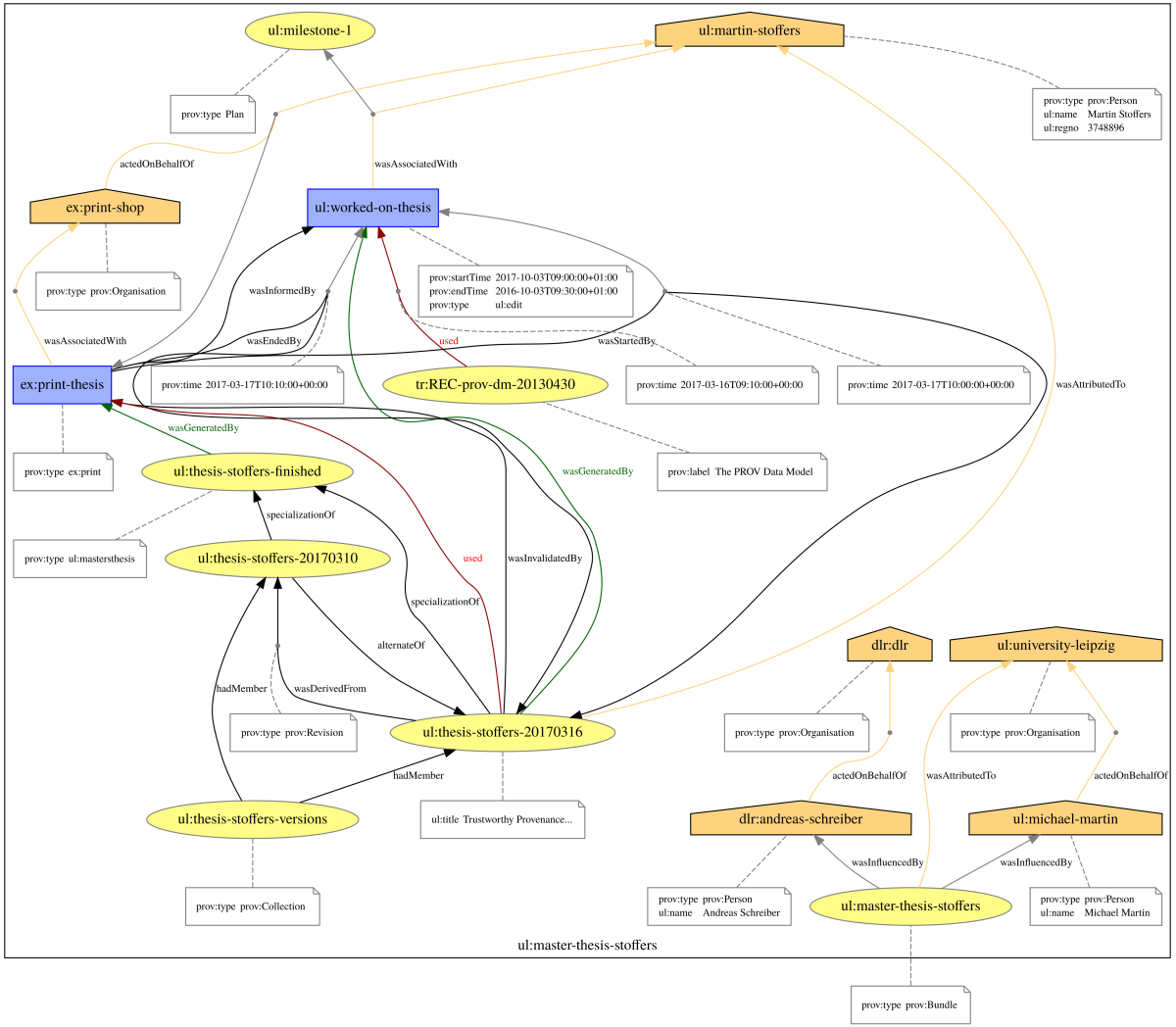


Figure A.8: Full PROV example as graph representation

Appendix B

Blockchains

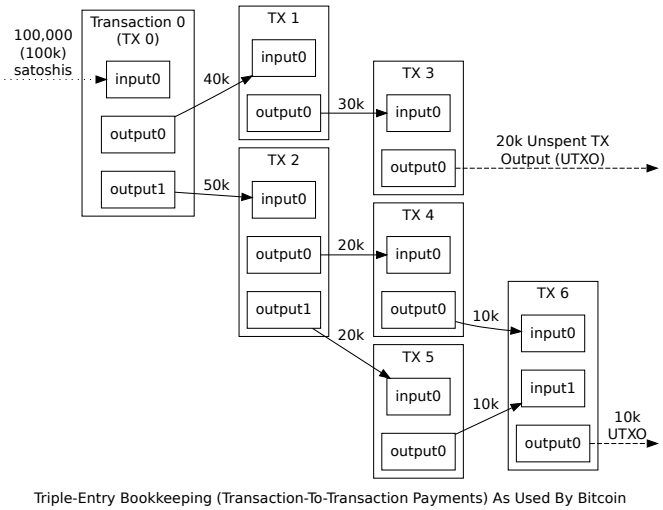


Figure B.1: Transaction Propagation [32]

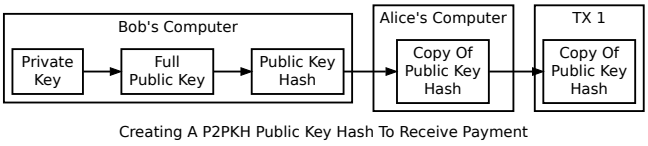


Figure B.2: Creating Public Key Hash [32]

Table B.1: Storage Type Comparison

Application	**Storage type**	**Size limit**	**Reliability**	**Tamper-resistance**	**Costs**
Bitcoin/Altcoins	OP_RETURN transaction output	80 bytes	+	+++	- -
Bitcoin/Altcoins	Coinbase transaction input	96 bytes	++	+++	- -
Ethereum	Account Storage	unlimited	+	+	-
Ethereum	Log Storage	unlimited	++	+++	-
BigchainDB	Account Storage	unlimited	+	-	+

Statement of Authorship

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.
Ich versichere weiterhin, dass die gedruckten und digitalen Ausführungen der Arbeit inhaltlich identisch sind.

Cologne, Thursday 30th March, 2017

Martin Stoffers