Keeping Data Inter-related in a Blockchain

Phani Chitti and Ruzanna Chitchyan(✉)

Department of Computer Science, University of Bristol,
MVB Building, Woodland Road, Bristol BS8 1UB, UK
{sc18092,r.chitchyan}@bristol.ac.uk

Abstract. Blockchains are gaining a substantial recognition as an alternative to the traditional data storage systems due to their tampering-resistant and decentralized storage, independence from any centralized authority, and low entry barriers for new network participants. Presently blockchains allow users to store transactions and data sets, however, they don't provide an easy way of creating and keeping relationships between data entities. In this paper we demonstrate a solution that helps software developers maintain relationships between inter-related data entities and datasets in a blockchain. Our solution runs over Ethereum's Go implementation. This is the first step towards a database management-like middleware system for blockchains.

Keywords: Blockchain · Data integrity · Referential integrity · Relations · Data access · Data stores

1 Introduction

Blockchains are increasingly recognised as an alternative to traditional database systems for transactional data and dataset storage. Yet, to be a feasible replacement to the currently predominating *relational databases* for the enterprise use, blockchains must also ensure that *integrity constraints*[1] hold for their stored data and for the interrelationships between such data entries. These *(data and referential) integrity constraints* ensure accuracy and consistency of data over its life-cycle and are important aspects of the design, implementation and usage of database solutions.

Data integrity refers to the overall completeness, accuracy and consistency of the stored data. *Referential integrity* ensures that data references spread across

[1] We note that some popular databases (e.g., NoSQL) do not address referential integrity constraints, which, we think, is one of the reasons of their slow penetration into the mainstream enterprise use.

This research is funded by the UK EPSRC Refactoring Energy Systems fellowship (EP/R007373/1).

© Springer Nature Switzerland AG 2019

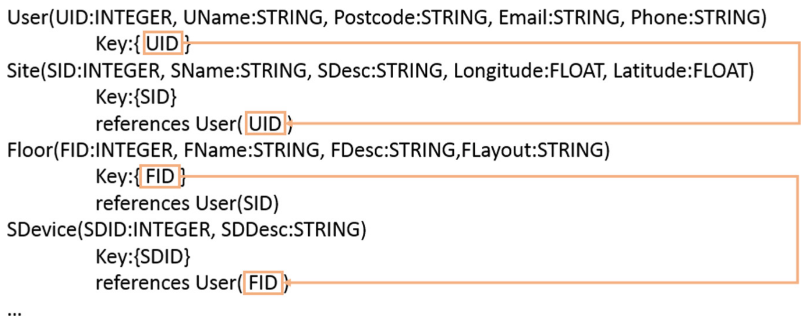
entities[2] are consistent. In a database, operations that modify the state of data (like inserting a new entry, updating or deleting a data record) must maintain the integrity constraints.

Presently blockchains don't provide an easy way for creating, maintaining, and using relationships between data entries, or for enforcing and utilising the integrity constraints. In this paper we demonstrate a solution that helps software developers maintain relationships between smart contracts, preserve these relations and enforce them during run-time. In short, our solution aims to provide to Ethereum blockchain users some of the functions that a traditional database management system delivers to its' users. Our solution runs over Ethereum's Go implementation.

The paper presents a motivating example for this problem in Sect. 2. The proposed architecture of the system and its components are explained in Sect. 3. Section 4 briefly discusses some related work, and Sect. 5 concludes the paper.

2 Motivating Example

To demonstrate the need for data and referential integrity, let us consider the example of a building occupancy analysis system [14]. This system collects and analyses large data sets about a building: its' sensors, equipment, energy consumption and occupancy information. Figure 1 shows a part of the conceptual schema for data representation in such a system, where inter-relations between entities are highlighted via linked boxes.



```
User(UID:INTEGER, UName:STRING, Postcode:STRING, Email:STRING, Phone:STRING)
          Key:{UID}
Site(SID:INTEGER, SName:STRING, SDesc:STRING, Longitude:FLOAT, Latitude:FLOAT)
          Key:{SID}
          references User(UID)
Floor(FID:INTEGER, FName:STRING, FDesc:STRING,FLayout:STRING)
          Key:{FID}
          references User(SID)
SDevice(SDID:INTEGER, SDDesc:STRING)
          Key:{SDID}
          references User(FID)
...
```

Fig. 1. Relational schema

Traditionally, a database management system provides means to implement such conceptual models within a database, as well as to ensure integrity when adding or maintaining their relevant data. Let us consider how relational and blockchain databases handle this.

[2] I.e., digital representations of the real world objects or concepts.

2.1 Integrity in a Relational Database

The *relational data model* [6,7] is built around the mathematical structure of *relation*. A *relation*, is a digital representation of an object or a concept (as defined in a schema) which is instantiated as a tabular construct (i.e., with columns for headings and rows for data entries). Constraints can be expressed over a set of columns to ensure data and referential integrity.

The Structured Query Language (SQL) CREATE TABLE command in Listing 1.1 creates a relation for the *Site* concept (shown in Fig. 1 conceptual schema).

```
CREATE TABLE IF NOT EXISTS Site(
SID INT NOT NULL, SName VARCHAR(255) NOT NULL,
SDesc VARCHAR(255) NOTNULL,
...
UId INT NOT NULL,
PRIMARY KEY(SID),
FOREIGN KEY(‘UId’) REFERENCES ‘User’(‘UId’)
                                ON DELETE CASCADE)
```

Listing 1.1. Create SQL Statement

the text that is highlighted in bold in Listing 1.1 indicates the data and referential integrity constrains over corresponding columns as explained below:

- **NOT NULL** construct prevents the corresponding column from accepting NULL values, thereby ensuring that a valid data entry always has a non-null value in this column.
- The **PRIMARY KEY** construct uniquely identifies each record in a table and is also used in data indexing.
- The **FOREIGN KEY** construct establishes relationships between Site and User tables, by referring and including the unique *UId* from *User* table into the *Site* table.
- The **ON DELETE CASCADE** construct deletes all corresponding records in the child table (Site) when a record in a parent table (User) is deleted, thus ensuring data consistency when change happens across relations.

The INSERT Statement in Listing 1.2 inserts data into *Site* table while referencing *UId* from *User* table.

```
INSERT INTO Site VALUES(1,‘‘Site−1”,‘‘First Site”,
                                  −2.774757,50.628323,1)
```

Listing 1.2. Insert SQL Statement

Since the database management system ensures the data and referential integrity constraints when updating data entries, the erroneous data entry is prevented. For instance, it is not possible to enter a NULL or a STRING value for *SID* attribute of *Site* table, as the respective constrains (as per Listing 1.1) reject such data entries. Similarly it is not possible to enter *Site* details for a User who is not present in *User* table, as the FOREIGN KEY('UId') constraint in Listing 1.1 ensures that the UId must be an existing unique ID in *User* table. The database management system performs corresponding actions for aforementioned integrity related constructs during insert, update and delete operations to ensure the data and referential integrity.

2.2 Integrity in a Blockchain

Ethereum blockchain is an immutable chain of interlinked blocks maintained in a network of untrusted peers. Ethereum blockchain can be used as a data store by modelling entities as smart contracts and deploying them to blockchain. Given the conceptual schema in Fig. 1, a sample outline for the contract representing the *User* entity (written in Solidity [9] language) is shown in Listing 1.3.

```
contract User {
    //Specify data Layout representing the
    //attributes of the entity
     struct UserDetail{
                uint16 UId;
                string UName;
                string Postcode;
                string email;
                string phone;
        }
    //mapping data structure used to store the Rows
    mapping(int => UserDetail) uDetails;
    //counter for key in mapping
    uint32 counter = 0;
    constructor() public { }
    //The Set methods inserts data row
    function SetData(address hash, uint16 uid, ...)
            public returns(string) {
        //Get the next key value by increasing the counter
        counter++;
       //Create a row using UserDetail Struct
       //Append the row to uDetails mapping using the counter as key
    }
    //The Get method return required data from
    //the mapping data store.
    function GetData(address hash) public view
                        returns (uint16 uid, ...) {
    ...
     }
    }
```

Listing 1.3. User Smart Contract

The text that is highlighted in bold in Listing 1.3 shows the constructs that set out the data layout, and functions for data insertion and access.

- **struct:** Solidity's *struct* construct specifies the data column layout for attributes along with their data types. The UserDetail struct defines the data layout for *User* as per the conceptual schema defined in Fig. 1.

- **mapping:** The *mapping* data structure specifies the data row layout for *User* entity; it stores data in key-value pairs. Alternatively the *event* construct can be used to store data. However, the drawback of using events is that the data cannot be easily accessed and marked when it is updated or deleted.
- **SetData:** the SetData function receives data from an application and inserts data into blockchain by appending a row in the *uDetail* mapping.
- **GetData:** many variants of the GetData function (as needed for the data requested by the caller) retrieve data from the *mapping* structure and return to the caller.

This approach has the following drawbacks:

- Uniqueness of a column: There is no straightforward way to ensure the uniqueness of an entity's "key" column. As a result, it is possible to insert two rows that have the same UId. The *mapping* uDetails then stores both records, as shown in Table 1, where user ID is duplicated.

Table 1. Mapping structure containing the inserted values for User contract

Key	Value
1	1,"ABC","BS8 1PT","xyz@bristol.ac.uk","745967262"
2	1,"DEC","BS9 3PT","abc@bristol.ac.uk","749857262"

- Referential integrity: To enforce referential integrity constraints (e.g., checking that a valid user with an ID '2' exists in the User contract, before inserting a data for ID '2' into Site contract), Ethereum requires use of a structured process [18]. So, for the above example, the Site contract will have to call the getData function of the User contract to check the legitimacy of ID '2'. The steps involved in this process are that:
  - The callee contract (User) must be deployed into blockchain beforehand (i.e., prior to Site calling it), to get its address.
  - The Application Binary Interface (ABI) of the deployed contract (User) must be specified in the caller contract along with the address of the deployed smart contract.
  - Public functions of the callee can be used via the ABI code and address.

  This is a rigid process and makes the modelling cumbersome (when compared to the simple SQL statement definition).

As a result, when the entities are modelled as individual contracts, without efficiently implementing data and referential integrity features, it is possible to have redundant, inconsistent and erroneous data.

Thus, below we present a solution that delivers (some of the) same features for the Ethereum users, that the database management systems deliver to the relational database system users. Our management system allows users to define relationships between smart contracts, preserves these relationships and ensures the referential integrity is maintained while performing operations on data. The relationships between smart contracts are represented as a Directed Acyclic Graph (DAG) and are stored in an underlying blockchain along with metadata.

3 Proposed Solution

The proposed system's architecture is presented in Fig. 2. Here the entities and their attributes (that were identified in the conceptual schema in Fig. 1) are modelled as smart contracts and deployed into the blockchain (Fig. 2, box a). The relationships between entities are captured in a directed acyclic graph (Fig. 2, box b) and will be handled by the Management System (Fig. 2, box c). Accordingly, the system has two main components: the data model, and the management system.

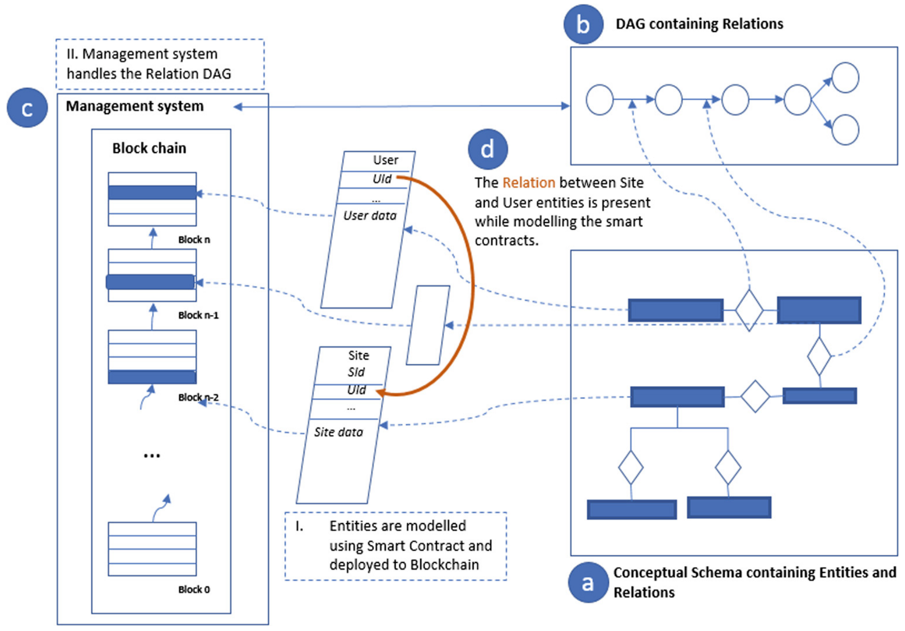

b
DAG containing Relations
II. Management system handles the Relation DAG
c
Management system
Block chain
Block n
Block n-1
Block n-2
...
Block 0
User
UId
...
User data
d
The Relation between Site and User entities is present while modelling the smart contracts.
Site
SId
UId
...
Site data
I. Entities are modelled using Smart Contract and deployed to Blockchain
a
Conceptual Schema containing Entities and Relations

Fig. 2. Architecture overview

3.1 Ethereum-Based Data Model

This data model uses the Ethereum smart contract as the data structure to model entities, and their attributes in a conceptual schema. The smart contract for management system's user entity defined in a conceptual schema is displayed in Listing 1.4, titled msUser.

The Listing 1.3 *User* smart contract is similar to the *msUser* in Listing 1.4 except for the *Detail* mapping definition. To achieve data integrity, the proposed system is using *Address* as a key in the mapping data structure.

```
        contract msUser {
         //Specify data Layout representing the
         //attributes of the entity
         struct UserDetail{
                uint16 UId;
                    ...
            }
       //mapping data structure used to store the Rows
        mapping(address => UserDetail) uDetails;
            constructor() public { }
        //The Set methods inserts data row
            function setData(address hash, uint16 uid, ...)
                                public returns(string)
            { ... }
        //The Get method return required data from
        //the mapping data store.
        function getData(address hash) public view returns
                    (uint16 uid, ...)
            { ... }
        }
```

Listing 1.4. User smart contract in Management System

Address is hash value calculated using the values of 'key' attributes that ensures data integrity. In the *User* contract, the *UId* is a key field. Hence the hash would be calculate on the value of *UId* and then would be inserted into *User* contract. For instance, to insert the below row into *msUser* contract, the value of *UId* field (1) would be hashed[3].

{1,"ABC","BS8 1PT","xyz@bristol.ac.uk","745967262"}.

After insertion, the mapping uDetails is as shown in Table 2. If an attempt is made to insert another row in which value of *UId* is 1, the insertion would fail as the mapping already contains this key. This ensures the uniqueness of the column and hence data integrity. In case of a multi-column key, a cumulative hash will be generated. Where an entity in a conceptual schema does not have

Table 2. Mapping structure containing the inserted values with Hash

Key	Value
0xabXn3f6d....2Ccd1P6LG	1"ABC","BS8 1PT","xyz@bristol.ac.uk","745967262"

[3] Hashing is used for creating a single-string key for establishing uniqueness, given that both single and multi-column references would need to be handled, as discussed below.

a key column (i.e., single column data set), the hash will be calculated from the value of the column and used as a key.

3.2 Management System

The Management system achieves the referential integrity and has two responsibilities: representing the relationships and ensuring the referential integrity among data entities. The following sections explain each of them in detail.

Representing Relationships

The Directed Acyclic Graph (DAG) data structure is used to represent the relationships among the entities. The DAG is defined as

$$Relations(DB) = <N, E>$$

Where N is a set of nodes relating to all entities in the DB (database) schema and E is a set of edges indicating relationships among entities. For the conceptual schema defined in building occupancy analysis example, N and E are defined as:

- N = {User, Site, Floor, SDevice, MultiSensor, SmartEquipment}
- E = {User–Site, Site–Floor, Floor–SDevice, SDevice–Multisensor, SDevice–SmartEquipment}

The graph is represented as an adjacency matrix. The elements of the matrix indicate the relationships between entities. The in-degree of an entity (number of incoming relationships) can be computed by summing the entries of the corresponding column, and the out-degree (number of outgoing relationships) can be computed by summing the entries of the corresponding row. The adjacency matrix in Table 3 indicates the relationships among entities in the aforementioned conceptual schema of Fig. 1. In the Boolean representation of adjacency (shown in Table 3), 1 represents a relationship between entities and 0 indicates absence of a relationship. The entities *User* and *Site* are related hence the corresponding element is represented with 1 in the Matrix. It can be seen from the matrix that the entity *Site* has an incoming relationships from User and an outgoing relationship to *Floor*.

Table 3. Adjacency Matrix representing the relations among entities

	User	Site	Floor	SDevice	MultiSensor	SmartEquipment
User	0	1	0	0	0	0
Site	0	0	1	0	0	0
Floor	0	0	0	1	0	0
SDevice	0	0	0	0	1	1
MultiSensor	0	0	0	0	0	0
SmartEquipment	0	0	0	0	0	0

Referential Integrity
The Management System ensures the validity of data references among different entities using the metadata about deployed contracts and relationships among them. The metadata stores information about entities and their attributes in predefined smart contracts. These predefined smart contracts are as follows:

- *SCRepository* contract stores the addresses of deployed smart contracts that represent the entities. This smart contract also provides functionality to obtain the address of the corresponding smart contract given an entity name and vice-versa.
- *SCEntityMetada* contact stores metadata about an entity, the data layout and other integrity related information. The data layout contains a list of columns and their data types in the order they are entered. This smart contract also provides functionality to obtain an entity's metadata given an entity name and vice-versa.
- *SCRelationDetails:* as discussed before, the management system stores the adjacency matrix representing the relationships among entities. The set of entities N will be stored as per the order they are arranged in the adjacency matrix. Keeping the node order is important as the relationships are defined based on this order. The set of relationships E is converted to a sequence of 1s and 0s, as shown below:

  $$\{E = 0,1,0,0,0,0,1,0,1,0,0,0,0,1,0,1,0,0,0,0,1,0,1,1,0,0,0,1,0,0,0,0,0,1,0,0\}.$$

  The SCRelationDetails smart contact stores relationship information among different entities and also provides functionality to obtain relationship data from an entity metadata given the address of an entity.

Figure 3 shows the sequence of actions taken by the Management System to ensure the referential integrity during a data insertion operation. The below row would be inserted into *Site* contract as the value for *UId* is present in *User* contract.

{2,"Site2","Second Floor",-2.773238,50.844838,1}.

Table 4 shows the data in the *Site* contract, after successfully inserting the row.

Table 4. Mapping structure containing the inserted values for Site contract

Key	Value
0xbdhe3a0d819....j2Lsd3D3AF	1,"Site2","Second Floor",-2.773238,50.844838,1

The below rows can not be added to *Site* contract, as the value provided for *UId* is not present in *User* contract, ensuring the referential integrity.

{1,"Site1","First Floor",-2.773838,50.883838,2}.

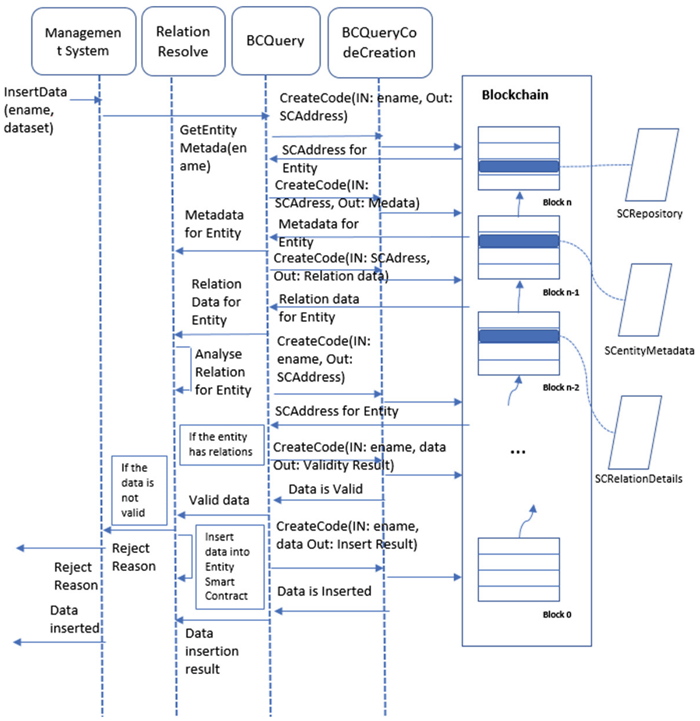


Fig. 3. Sequence diagram

4 Related Work

Since emergence of Bitcoin [17] a wide range of research and development activities is under way on using blockchain as a data storage mechanism. A small selection of such related work is presented below.

One area of research is in transferring properties of blockchain to centralised databases and/or database properties to blockchains. For instance, BigchainDB [4] integrates properties of a blockchain into a database, while using an Asset as a key-concept. Asset represents a physical or digital entity which starts its' life in BigchainDB with CREATE Transaction and lives further using TRANSFER Transaction. Each Asset contains Metadata to specify details about the Asset. The life-cycle of a BigchainDB Transaction is described in [5]. Each node in BigchainDB will have a MongoDB [15] instance and will maintain the same set of data. The Tendermint [19] is used as consensus protocol, which ensures the data safety even if 1/3 of nodes are down in the network. Traditional database features, like indexing and query support are carried out through the underlying MongoDB.

Mystiko [3] is another attempt in the same direction where Cassandra [2] is used in every node in the network as a database. Cassandra is a No-SQL column store database and facilitates the features such as full text search and indexing options for Mystiko users.

Another focus area is creating customised private/permissioned blockchains. Hyperledger [1] is a permissioned open source blockchain and related tools, started in December 2015 by the Linux Foundation. Quorum [16] is a permissioned blockchain platform built from the Ethereum codebase with adaptations to make it a permissioned consortium platform. Ethereum code base is modified with private transactions, consensus (Proof-of-work to voting system). Quorum creators increased the transaction rate. The Energy Web Foundation(EWF)'s Energy Web is an open-source, scalable Ethereum blockchain platform specifically designed for the energy sector's regulatory, operational, and market needs [8].

Hyperledger [1] is a permissioned blockchain system focused on scalability, extensibility and flexibility through modular design. Different consensus algorithm can be configured to this solution (e.g., Kafka, RBFT, Sumeragi, and PoET) while smart contracts [10] can be programmed using a platform called Chaincode with Golang [11]. Hyperledger uses Apache Kafka to facilitate private communication channels between the nodes. It can achieve up to 3,500 transactions per second in certain popular deployments. Work to support query [12] and indexing [13] on temporal data in blockchain using Hyperledger is also underway.

In a similar vein, our work looks at integrating the well established relationship management mechanisms of the relational databases with the blockchain infrastructure.

5 Conclusions and Discussion

In this paper we have presented an initial implementation of a middleware that preserves relationships among entities while storing data in a blockchain. This middleware has two components, the Ethereum-based data model that supports users in modelling entities as smart contracts and a management system that maintains relationships between entities.

However, as this is an initial proof-of-concept implementation, it has a number of limitations, such as:

– Dependency on *solc* tool, which is used to compile smart contracts to create ABI, BIN and .GO files. The *solc* changes whenever the solidity specification changes. Currently, our middleware will have to change to adapt to such tool changes.
– Time delay in blockchain response, caused by the need to carry out additional validation activities while carrying out the usual insert/update operations. This could be unacceptable, particularly when working on processing high frequency data.
– Storing temporary files: while compiling a smart contract, a set of files (.abi. .bin and *.go) are generated which must be available to create GO programs that interact with the blockchain. Over the lifetime of the system these files would grow in number and size, which could cause storage considerations.

Our immediate future research will focus on addressing the above pointed limitations, always working towards making blockchains a more widely usable database solution.

References

1. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: EuroSys, NY, USA, pp. 30:1–30:15. ACM (2018)
2. Apache: Apache Cassandra. http://cassandra.apache.org/. Accessed 14 Feb 2019
3. Bandara, E., et al.: Mystiko-blockchain meets big data. In: IEEE Big Data, pp. 3024–3032, December 2018
4. BigchainDB: Bigchaindb..the Blockchain Database. https://www.bigchaindb.com/. Accessed 14 Feb 2019
5. B. Blog: Lifecycle of a BigchainDB Transaction - The BigchainDB Blog. https://blog.bigchaindb.com/lifecycle-of-a-bigchaindb-transaction-c1e34331cbaa. Accessed 14 Feb 2019
6. Chen, P.P.-S.: The entity-relationship model-toward a unified view of data. ACM Trans. Database Syst. **1**(1), 9–36 (1976)
7. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
8. EWF: The Energy Web Blockchain-Energy Web Foundation. Accessed 20 Feb 2019
9. E. Foundation: Solidity - Solidity 0.4.24 Documentation. https://solidity.readthedocs.io/en/v0.4.24/. Accessed 14 Feb 2019
10. E. Foundation: White paper.ethereum/wiki wiki github. https://github.com/ethereum/wiki/wiki/White-Paper. Accessed 14 Feb 2019
11. Google: The Go Programming Language (2019). https://golang.org/. Accessed 13 Feb 2019
12. Gupta, H., Hans, S., Aggarwal, K., Mehta, S., Chatterjee, B., Jayachandran, P.: Efficiently processing temporal queries on hyperledger fabric. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1489–1494, April 2018
13. Gupta, H., Hans, S., Mehta, S., Jayachandran, P.: On building efficient temporal indexes on hyperledger fabric. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 294–301, July 2018
14. Ioannidis, D., Tropios, P., Krinidis, S., Stavropoulos, G., Tzovaras, D., Likothanasis, S.: Occupancy driven building performance assessment. J. Innov. Digit. Ecosyst. **3**(2), 57–69 (2016)
15. MongoDB: Open Source Document Database — MongoDB. https://www.mongodb.com/. Accessed 14 Feb 2019
16. Morgan, J.: Quorum — JP Morgan. Accessed 20 Feb 2019
17. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
18. Peh, B.: Calling a Function Another Contract in Solidity. https://medium.com/@blockchain101/calling-the-function-of-another-contract-in-solidity-f9edfa921f4c. Accessed 06 Mar 2019
19. Tendermint: Blockchain Consensus - Tendermint (2019). https://tendermint.com/. Accessed 14 Feb 2019