



University of
Zurich^{UZH}

Design and Implementation of a Database-to-Blockchain Data Gathering Solution for Cheese Tracking

David Diener
Zurich, Switzerland
Student ID: 19-733-179

Supervisor: Jan von der Assen, Eder J. Scheid, Christian Killer
Date of Submission: August 25, 2022

Abstract

Globalization shaped our markets and the world that we live in today as nothing ever else has. New opportunities presented themselves, remote markets got opened up and supply chains got more complex. With increased complexity came several challenges, such as motivation for exploitation, costs for trusting parties, and abuse of market power, which needed and still need to be tackled. The introduction of the Blockchain technology in 2008 made a step in this direction, such as immutability and higher security, which can be profited from, to build a more secure Supply Chain. Tête-de-Moine a Swiss cheese producer, the University of Zurich, Fromarte, and Agroscope found a similar need to bring more trust along the value chain of its production. Accordingly, a Proof of Concept, for a part of the overall CheeseChain project was established in this thesis. The implemented solution mirrors the database from the Digital Quality Management System of Fromarte on the host and pushes relevant information onto a private blockchain. With an API, combining the Blockchain and the host, the integrity of the files in the database can be assured. The solution was assessed through several performance tests, which led to the conclusion that the BC interaction takes up to 80% of the first iteration. This brought the question of how salable the solution is. Tests showed that the time it takes to fetch a form follows a linear scale, which makes the scalability anticipatable.

Zusammenfassung

Die Globalisierung prägte unsere Märkte und unsere Welt wie nichts anderes zuvor. Neue Möglichkeiten entstanden, ferne Märkte wurden erschlossen und Lieferketten wurden komplexer. Mit steigender Komplexität kamen neue Herausforderungen: Motivation des Ausnutzens, steigende Kosten für vertrauenswürdige Partner und Missbrauch von Marktmacht. Das Aufkommen der Blockchain Technologie im Jahr 2008 machte einen Schritt in diese Richtung von mehr Sicherheit, da sie wertvolle Eigenschaften mit sich bringt, die für eine sichere Lieferkette gebraucht werden. Tête-de-Moine ein Schweizer Käseproduzent, die Universität Zürich, Fromarte und Agroscope haben einen ähnlichen Bedarf festgestellt, mehr Vertrauen in die Wertschöpfungskette ihrer Produktion zu bringen. Entsprechend wurde eine Machbarkeitsanalyse für ein Teil des CheeseChain Projekts in dieser Arbeit durchgeführt. Die implementierte Lösung spiegelt die Datenbank des digitalen Qualitätsmanagementsystems von Fromarte auf den auf den Host, auf dem der Code läuft, der die wichtigsten Informationen dann weiter auf eine private Blockchain schreibt. Kombiniert mit einer API, die die beiden Systeme zusammenführt, kann Integrität der Formulare in der Datenbank versichert werden. Die implementierte Lösung würde auf ihre Leistung überprüft. Die Tests zeigten, dass die BC Interaktion mehr als 80% der Zeit beansprucht. Das brachte die Frage nach der Skalierbarkeit hervor. Weitere Tests zeigten, dass die Zeit, die es braucht Formen von der DigitalQM zu laden, einer Linearen Skala folgt, was die Skalierbarkeit antizipierbar macht.

Acknowledgments

First, I would like to express my special thanks to my supervisor, Dr. Eder John Scheid for his pervasive support and competent insight into the topic. During the writing of the thesis, I continuously got his professional review, for which I'm extremely thankful. Furthermore, I want to express my gratitude to Prof. Dr. Burkhard Stiller for the opportunity to be part of the CheeseChain project, and thank the members of the Communication Systems Group and the Department of Informatics at the University of Zurich. Finally, I want to thank my brother, girlfriend, and friends, who always offered their assistance.

Contents

Abstract	i
Acknowledgments	v
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Outline	2
2 Background	5
2.1 Blockchain	5
2.1.1 Blockchain Technology	5
2.1.2 Types of Blockchains	9
2.1.3 Private Blockchain implementations	11
2.1.4 Smart contracts	11
2.1.5 Different eras of Blockchain	12
2.2 Supply Chain Management (SCM)	12
2.2.1 Supply chain tracking	12
2.2.2 Problems with traditional supply chain management	13
2.3 Cheese-Chain Project	13
3 Related Work	15
3.1 Supply Chain re-engineering	15
3.2 Supply Chain Tracking	16
3.2.1 Ethereum-based Projects	16
3.2.2 Hyperledger-based Projects	17

4	Design and Implementation	19
4.1	Design	19
4.1.1	Data Gathering	20
4.1.2	Working with the BC	21
4.1.3	On- off-chain data storage	23
4.2	Implementation	23
4.2.1	Data Gathering	23
4.2.2	Data Parsing	25
4.2.3	Data Synchronization	27
4.2.4	Interaction with the private BC	30
5	Evaluation	33
5.1	Relation to the CheeseChain Project	33
5.2	Performance Testing	34
5.2.1	Run-time	34
5.2.2	Scalability	36
5.3	Limitations	37
5.3.1	Code limitaitons	37
5.3.2	Query limitations	37
6	Summary, Conclusion, and Future Work	39
6.1	Conclusion	39
6.2	Future Work	40
	Bibliography	40
	Abbreviations	45
	List of Figures	45
	List of Tables	47

<i>CONTENTS</i>	ix
A GraphQL queries	51
B Installation Guidelines	55
B.1 Set up	55
B.2 Run	55
B.3 Synchronization	56
C Contents of the ZIP	57

Chapter 1

Introduction

Over the last two decades, Blockchain technology has gained popularity in different areas of application. It all started in 2008 when the Bitcoin white paper was published and the Bitcoin was created a year later [2]. The rise of Bitcoin has inspired the creation of numerous other BC platforms (*e.g.*, Ethereum in 2015 [1]). Besides cryptocurrencies, the new platforms focused on integrating further functionalities such as Smart Contracts (SC) [42] or Decentralized applications (DAPPS).

Contemporaneously, globalization evolved further and *Value Chains* got more complex. Trust relations between entities take time, need reputation, or cost money. BCs new functionalities opened doors to utilize immutable, decentralized, and safety properties for different applications. Through the outsourcing of liabilities from entities of the value chain onto the BC, issues like security can be redistributed to the BC. This concept has found support in the food industry to ensure food safety [4, 45, 36].

Tête de moine a Swiss cheese producer, Formarte the umbrella organization of Swiss cheese dairies, Agroscope the Swiss center of excellence for agricultural research, and the Communication Systems Group (CSG) at the University of Zurich have started a similar project, to improve transparency and trust along the Tête de Moine value chain (called CheeseChain). A private BC is set up to track the changes in the value chain. Due to companies' data privacy and regulations, a project like this is often not suitable to operate on public BCs. Moreover, private BCs are regarded to be more suitable for Business-to-Business (B2B) applications [9].

This thesis serves as a Proof-of-Concept (PoC) for the CheeseChain project. A solution has been implemented to store fields of relevance from the Digital Quality Management System of Fromarte onto the host and the BCs. This includes the gathering of the Data from the database, extracting the relevant variables from the fetched data, and pushing it onto a private Blockchain.

1.1 Motivation

Research has gone into measuring the performance of BCs and comparing BCs to one another. Furthermore, several detailed projects have been implemented in the food industry. This thesis focuses on a smaller, more practical implementation, with on and off-chain data storage. Splitting the storage into on and off-chain data storage brings the following benefits:

- **Data privacy:** Sensitive data will not be published onto a public BC;
- **Fewer transaction costs:** Compared to when working exclusively with public BCs;
- **Security:** The solution exploits the properties of the BC technology, which makes the hybrid solution more tamper-proof;
- **Network preservation:** Not as many transitions, compared to when everything is stored on public BCs;
- **No need for a third party:** Through the BC solution, the need for a third party regulator gets redundant;

The implemented solution not only offers benefits for the participants along the value chain but also to the end consumer. As the importance of food safety and proof of origin increases, so does the motivation for forgers. This solution may bring clarity to consumers, as they can trace the different production steps through the BC and verify the authenticity of products themselves.

1.2 Thesis Outline

This Thesis is structured as follows:

1. Chapter 1 includes the introduction of the thesis' background and a short presentation of the topics.
2. Chapter 2 serves as an introductory chapter, and seeks to establish a common knowledge base, emphasizing BC technology, deployment types, and the context of the CheeseChain project.
3. Chapter 3 focuses on associated work done in the field of supply chain re-engineering, supply chain tracking through BCs, and on- and off-chain supply chain tracking, focusing on private BC implementations.
4. The main Chapter 4 includes the solution's design (Section 4.1) and implementation (Section 4.2). Through several illustrations, the design of the framework of the solution is described. The implementation section consists of several elaborated code snippets explaining the inner workings of the solution.

5. In Chapter 5, the solution is challenged and its performance will be evaluated. Secondly, the thesis is assessed from the perspective of the whole CheeseChain project. Thirdly, this chapter details the challenges faced during implementation.
6. Chapter 6 concludes this thesis by presenting its outcomes and future work.

Chapter 2

Background

This Section provides a description of the necessary elements to understand the technical background of this bachelor thesis, including details of Blockchain (BC), the management of supply chains, and the CheeseChain project.

2.1 Blockchain

The concept of BC started in 2008, when an anonymous author with the pseudonym Satoshi Nakamoto published a “handbook” for the creation of Bitcoin, called “Bitcoin: A Peer-To-Peer Electronic Cash System”[29]. Since then, the popularity of BC and its application area has evolved drastically in cryptocurrencies, industries like finance, agriculture[44], or in construction [48].

Essentially, BCs are a type of Distributed Ledger Technology (DLT), where all records are stored decentralized and digitally. Due to its properties, BC does not rely on a single source of truth, unlike conventional databases, which makes it less prone to malicious attacks defrauding the system [43].

2.1.1 Blockchain Technology

BC is a distributed peer-to-peer (P2P) data structure, which is replicated and shared amongst all members of a network. The following sections detail aspects of said technology.

Peer-to-Peer (P2P)

Decentralization and the P2P structure go hand in hand with BC technology. Basically, it is a network label, in which each participant operates as a client and server [50]. P2P applications do not depend on a centralized server to store and distribute information. This brings several advantages. Transactions are more accessible, portable, and the peers,

also known as nodes in the BC world, are in control [11]. Furthermore, decentralized systems have a significant impact on efficiency, reliability, and scalability [35][50].

Distributed Ledger Technology (DLT)

Distributed ledgers are data structures shared among multiple computer devices. The DLT is the underlying technology, that is permitting, storing, and updating ledgers in a decentralized fashion [5]. It consists of three main properties:

1. A data design that allows capturing of the current ledger state.
2. A collective communication language to alter the ledger's state.
3. A mechanism to reach consensus among all participants, on which transaction to accept by the ledger and in which order.

Architecture of a Blockchain

BCs are sequences of blocks, holding lists of transaction records. The chain grows by nodes appending a new block to the chain. The validity and authenticity of new blocks are inspected by other nodes. The mechanism of the validation will be covered in the next sub-chapter.

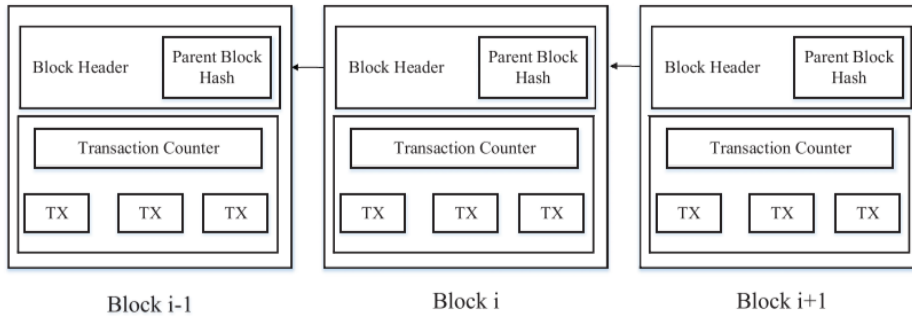


Figure 2.1: Linked Blocks forming a Blockchain[51]

A block holds several components. The most obvious one is the hash of the parent block, which forms the link between the blocks, hence forming a chain of blocks, or rather a Blockchain - see Figure 2.1.

As shown in Figure 2.2, a Block consists of two main parts: the header and the body. The header includes[51]:

- Block version: holds the validation rules.
- Merkle tree root hash: holds the hash value of all transactions inside the block.

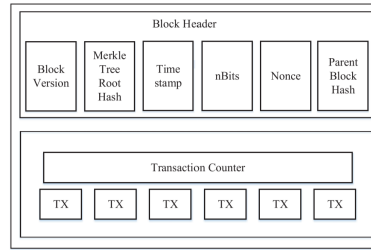


Figure 2.2: Inner structure of a Block[51]

- Timestamp: current time in seconds
- nBits: the target threshold of a valid block, which need to be met to validate a block.
- Nonce: A number used during the calculation of the target hash, further explained in 2.1.1.
- Hash to the parent's block: this is a 256-bit hash value pointing to its parents block

Apart from the first block, the so-called Genesis block, every block of the BC holds such information. The Genesis block cannot carry information from a parent block, since it has none. This list may vary depending on the BC, and its individual properties.

Nodes and handling of transactions

Every node has an equivalent copy of the BC (ledger), and can verify or interpret the state of the BC [3] [10]. Not all nodes are involved in the validation of new blocks. If they are, they are called miners. Depending on the consensus mechanism (see Section 2.1.1), miners are commissioned with different computing tasks to approve a new block and hence add to the BC.

Transactions are propagated through the network to validating peers i.e. the miners. The mining nodes can then apply the BCs consensus mechanism. Once a mining node considers a block as valid, the information is broadcasted through the network, so all nodes can update their local replica. Lastly, the new block is confirmed in the final version of the ledger, which makes it immutable [5].

As stated above, transactions (*e.g.*, cryptocurrency, smart contracts) are processed through nodes. To be able to interact with the BC each user owns a pair of private and public keys [10].

Cryptography

Essentially, cryptography protects data in a non-trusting environment from being tampered with. Hashing of data brings several benefits. Not only will the data get compressed,

which makes the system faster and transactions less expensive, but it also happens in an irreversible fashion. The latter is vital to ensure the integrity of the data inside the network [5]. BC technology combines asymmetric cryptography with hashing and digital signature, which form the pillars of its security [30]. This process happens in three phases:

1. Key-pair generation: Each user of the BC generates a private key, to sign their transaction with and a public key, so the receiver is able to verify the authenticity, origin, and integrity of the data [5].
2. Signing: The sent data is hashed by the sender and the digital signature is generated with the private key. The signed hash gets sent collectively with the encoded origin data [5].
3. Verification: The received data is decoded with the sender's public key and matched with the re-computed hash of the raw data[5].

Consensus mechanisms

By its nature, BCs are networks of untrustworthy parties, which makes it challenging to reach consensus among the nodes. In [48], the authors use the analogy of the Byzantine Generals problem. A group of military generals surrounds an enemy's city. The attack would only be successful if all the generals attacked the city. Hence they have to reach an agreement on the strategy, to attack or to fall back. BCs also need to find consensus among the nodes else they fail their purpose [47]. As a distributed system, a BC has no central point of trust that could ensure the correctness of ledgers in different nodes. This brings the need for a mechanism to reach a unity of truth [21].

Due to the decentralization of BCs, it is possible that two nodes mine a block synchronously which leads to forking. Forks can be compared to branches on a tree or in GitHub. Different BCs resolve that problem in a distinct manner [26][46]. The following section detail selected consensus mechanisms.

- Proof of Work (PoW): This consensus mechanism is best known from the Bitcoin network. Due to the decentralization, a node has to be selected to record new transactions. If this were done through randomization, the system would be prone to attacks. To ensure safety inside the network hurdles need to be set up to make sure the node is not likely to attack the network. The main hurdle in this mechanism is the complexity and costs of the computing calculations a node needs to perform to mine a new block. A node mining a new block needs to combine the hash value of the last mined block, the transactions to be validated in the block, the public key of the miner and a nonce to a string (s), which is used as an input of the hash function $H(s)$ [19]. The goal is to get a hash value that is equal to or smaller than a given value, so the miners need to change the nonce frequently to get a new and hopefully correct output. As soon as a node reaches the aimed value, it broadcasts the new block, which then all other nodes need to confirm according to the hash value [51]. The PoW mechanism makes sure that honest parties obtain control of

the chain, as it will be quite unlikely that a malicious party will have control of so much computational power to get control. Nevertheless the mechanisms are still vulnerable to some attacks, which will be covered in the next Section [19].

- Proof of Stake (PoS): The main difference from PoS to PoW lies in the decision on who has the right to mine a new block. Unlike in PoW, miners in PoS do not have to compete against each other. The philosophy of PoS is that users with more currency inside the system have higher stakes to make sure that the system works truthfully, so they will be selected more frequently to mine a block. This selection is quite biased towards users with lots of currency inside the network.
- Delegated Proof of Stake (DPoS): In DPoS the voting is the other way around. Before a block is published, users vote for nodes to have the right to publish new blocks. Obviously, users with higher stakes, have an increased voting power [30].
- Byzantine-Fault-Tolerance (BTF): BTF and PoS are quite similar. In BTF a node is chosen randomly, while staked nodes need to confirm the new block by voting, which gives every node a “voice”. Of course, it is assumed that honest nodes never accept a malicious block [19].

Safety, problems and possible attacks

The mentioned consensus mechanisms above aim to protect the honest nodes from bluff transactions and malicious actions, but some risks remain in the systems:

- 51% Attack: This is only possible in the PoW mechanisms. If a miner has control over 51% or more of the computing power inside a network, he/she is able to control it [21].
- Double spend: This problem mainly affects cryptocurrencies. Double spending means that a user is able to spend a coin twice. This is a complication many digital money systems face, but most BCs are designed to prevent such flaws [30].
- Nothing at stake: This problem may occur in some PoS mechanisms. If multiple forks exist in a BC, staked users may capitalize on such situations to “mine” blocks on every existing fork to increase their odds of financial rewards from transaction fees [30].

2.1.2 Types of Blockchains

Types of BCs can be categorized along two axes: Permission/permission-less and public/private. To categorize the types even further, we need the following definitions: *Read* permission means that all transaction from a chain is visible to every node in the network. *Write* on the other hand means if a node is able to modify or update the state of the BC [38].

Public Blockchains

In public BCs everyone in a network has the right to read past transactions. The only difference is that in *permissioned public BCs* the writing on the BC needs to be approved. This is not the case in *permission-less BCs*, where everyone is able to perform write operations [38].

Private Blockchains

Unlike in public BCs, read/write permission needs to be granted. If you are inside a network (*e.g.*, inside an organization), where permission is granted by a white-list, then we are talking about *private permission-less BC* [10]. If access needs to be granted by a central authority, we are in the field of *private permissioned BCs*[38].

Since private BCs operate in closed networks, they have a higher transaction processing rate. Thus less time is required to mine a new block, which lets the BC grow quicker [17]. As permission needs to be granted or one needs to be in a private network, identities of nodes are known[10], which makes changes possible if all nodes agree. In contrast, if sensitive information is leaked onto a public BC it is almost impossible to roll back[48].

The closed nature of private networks does not automatically shield them from malicious actors. If a user has bad intentions and gains access to the private network, it is easier to obtain control for the attacker due to the limited amount of nodes[48].

Consortium Blockchains

Unlike private BCs, consortium BCs are somewhat distributed, but the actors or nodes inside the network are still known. Similar to private BCs, a node must be verified by a pre-set node [49]. Since more than one Party is involved, this deployment type suits environments where more than one organization is reliant on adding transactions or getting information from the chain (*e.g.*, Banks, federal organizations or, supply chain tracking systems) [23].

Hybrid Blockchains

Hybrid BCs combine the two deployment types. Organizations can make use of the best properties from both the private and public BC. By combining the two, it is possible to exploit the private BCs features like high performance, security, and complete auditability, whilst having the option of verifying transactions on the public BC (*e.g.* through smart contracts) for transparency and verifiability[8].

2.1.3 Private Blockchain implementations

As a private BC is included in the solution of this thesis, two of the most advanced and most used BCs are introduced below.

Hyperledger

Hyperledger is an open source collaboration of BC technologies launched by the Linux Foundation [24]. Hyperledger is not a unique BC technology, like Bitcoin, but rather a collection of technologies, built for enterprise use cases. Each framework has its own specific advantages in different environments.

For example, Hyperledger Fabric is the most advanced Hyperledger Framework, as it originated from IBM. It is designed to be a company-oriented BC. Due to its modular architecture, the framework is highly modifiable. The advantages of the Hyperledger Fabric include identity management, channel privacy (a private “subnet” to communicate two or more members to perform private and confidential transactions), and chaincode to make use of smart contracts. As Hyperledger is optimized for commercial use, the system distinguishes between peers customers and the organization that owns it [24].

Ethereum

The public Blockchain Ethereum is a community-run technology powering the cryptocurrency ether (ETH) and thousands of decentralized applications [16]. For purposes like testing or when dealing with sensitive information, a private local single instance deployment is better suited, as nothing will be exposed to a public BC. Ganache is one of the most popular local Ethereum simulation programs, that can be interacted with, without any other participants [1].

If there is a need for several nodes when working with multiple participants, a full Ethereum deployment inside a local network is also possible. Through the deployment of *i.e.*, Geth (go-Ethereum), the private BC can be set up.

2.1.4 Smart contracts

The computer scientist Nick Szabo introduced the idea of smart contracts (SC) in 1994. He suggested embedding contracts into code, which could then be self-enforced. This would make the need for trusted parties (*e.g.*, notaries and banks) redundant [10].

His idea can be translated into the world of BCs. SC can be described as agreements or contracts of any kind adapted into scripts, which then are saved inside a block of a chain[42]. Once deployed, SCs are immutable, so the code cannot be changed. This implies that SCs are also autonomous and require no maintenance once deployed.

SCs are triggered to execute a contract, but only if all conditions of the SC are met[43]. It is possible that one condition is a different SC, which itself has dependencies or other conditions. Through oracles, SCs are able to "communicate" with the real world (*e.g.*, if the price hits X do this). This bridge is bi-directional, meaning that an SC can be triggered through something in the real world or the other way around[40].

2.1.5 Different eras of Blockchain

In this section, the different eras of the BC technology will be addressed. First, we start with BC 1.0, which covers what is most known in the public eye - currency. Continuing with BC 2.0, covering Smart Contracts. And finally, the application side in the BC 3.0 section, will be examined.

Blockchain 1.0 (Currency): The first BC was introduced with the upbringing of Bitcoin and the underlying technologies: Blockchain, Protocols (Programs performing transactions), and Cryptocurrencies. With said technologies and eWallets, transaction fees could be lowered, compared to transactions on the financial market [43].

Blockchain 2.0 (Contracts): Unlike BC 1.0, where the focus lay on currency and its transactions, BC 2.0 focuses on the decentralization of markets on a broader scale. Concepts like smart property (*e.g.* ETFs), smart contracts and decentralized storage are part of BC 2.0 [43].

Blockchain 3.0 (Applications): BC 3.0 takes it even further than BC 2.0, as it focuses on every aspect of all human endeavors. It aims to reorganize activities in every aspect of life, so they get faster and more efficient.

For example, combining BC with big data, processes, or predictions of reality could be turned into actions with BC technology. There is a large area where automation could be achieved through big data in combination with smart contracts (*e.g.*, FinTech industry).

2.2 Supply Chain Management (SCM)

The following quote from [27], introduces the concept of SCM: *In modern business management, individual businesses cannot compete as independent entities but rather as active members of the wider supply chain involving a network of multiple businesses and relationships.*

2.2.1 Supply chain tracking

The tracking and tracing of the supply chain allows you to track products from start to finish. This brings quality control to a new level. Due to strong globalization and outsourcing of production steps, the maintenance and overview of the supply chain have become progressively more complex in several industries [41][31]. Moreover, the authors

of [31] propose that the tracking and tracing of the supply chain data may be used to generate Key Performance Indicators (KPI), which is crucial in a competitive market.

2.2.2 Problems with traditional supply chain management

To maintain a manual supply chain with human intervention is tedious, time and cost-intensive, insecure, vulnerable to errors, and does not guarantee security throughout the supply chain. Furthermore, natural disasters, political policies, societal norms, and even attitudes of SCM stakeholders may negatively impact the supply chain [6]. This brings the need for a modernized SCM, which will be elaborated in Section 3.

2.3 Cheese-Chain Project

As this thesis is a sub-project of the CheeseChain project by the University of Zurich (UZH) in cooperation with Agroscope, Tête-de-Moine, and Fromarte, the CheeseChain project needs to be introduced [12]. The goal of the project is to increase transparency and trust along the Tête-de-Moine value chain. To achieve this goal, a BC will be set up to store the biological data (*i.e.*, laboratory data from PCR-analysis) of the supply chain combined with a partner-specific database. The BC allows the implementation of an automated detection system for fraudulent concerns, which leads to a more efficient sampling by the trademark owner. Along with the trademark owner, the authorities, suppliers, and end-consumers will also benefit from said solution [12].

Fromarte, the parent organization of the Swiss cheese dairies, considers CheeseChain as their future quality management software and values progress in the tracking of the supply chain in contrast to traditional production. The following benefit should result from the implementation of the CheeseChain [18]:

- Authentication of cheese should be automated, to identify forgeries.
- Increased food safety
- More efficient customs handling
- More accessible provisioning of product information for the end-user and authorities
- Real time monitoring of the supply chain

Stakeholders of the CheeseChain project are institutions or organizations that are affected or take part in the project.

As an investor, Innosuisse has the power of co-decision, which makes it a so called invested stakeholder. The primary stakeholders take part in the execution. In this case, Agroscope, Tête de Moine, Fromarte, and UZH are invested Stakeholders of the CheeseChain project. Cheese dairies are secondary stakeholders, as they are influenced by the project. Lastly, the consumers are the tertiary stakeholders since they use products or services [18].

Chapter 3

Related Work

With the rise of BCs, the field of its application has diverged a lot (as stated in Section 2.1.5). As it has also been widely explored that there are several drawbacks with centralized databases, the potential of the BC technology in the tracking of the supply chain has been found to be useful [48] [31] [36]. In the context of this thesis, a private BC will be set up, which is why the focus in the following chapters lies on private BCs in supply chain tracking.

3.1 Supply Chain re-engineering

Since the transition from a traditional supply chain(*i.e.*, paper trails, trust relationships between entities, and centralized point of trust) to a decentralized P2P solution is expensive and time-consuming, there is a need for an updated supply chain structure [22].

Conceptual Framework

Figure 3.1 illustrates a typical supply chain, consisting of material flow and information flow. In the past, several steps had to be included (*i.e.*, E-Mails correspondence and web-based services like ERP systems) to track the supply chain thoroughly. This brings the downside of potential delays, proneness to human errors and is overall labor intensive [39].

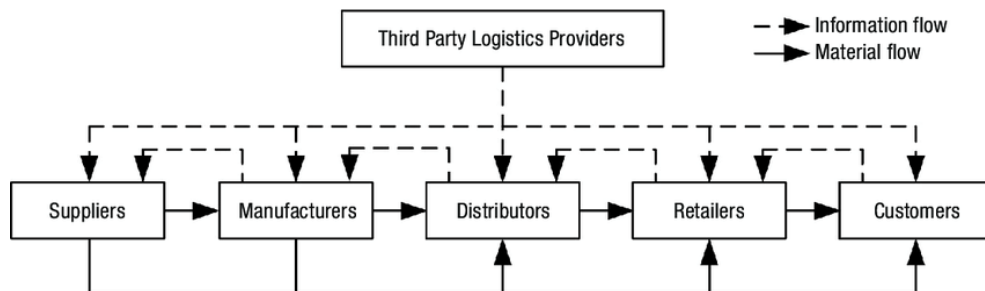


Figure 3.1: Traditional Supply chain process [25]

When re-engineering the overall structure of a supply chain, all contracts and communication that takes place into different stages of the supply chain need to be modeled in three sub-processes inside smart contracts, running on a BC network [39]:

1. The transaction process: A data structure to record and track key variables (*e.g.*, user ID, order Time, Shipping status, product ID, and quantity).
2. Payment process: This process is designed to avoid incomplete payments. Since there is no central authority in said concept, the handling of transactions and guaranteeing their completeness is of importance.
3. Data accessing process: Integrating on- and off-chain databases is crucial at this stage. Communication between the two can be achieved through smart contracts.

3.2 Supply Chain Tracking

Through globalization and distribution of production, a modernized approach to safety quality, and ubiquity is required. Several new approaches have been proposed regarding the incorporation of BCs into the supply chain. In this section, the focus lies on different solutions from various industries.

3.2.1 Ethereum-based Projects

In [36] the authors propose a tracking system for the soybean supply chain, based on the public BC *Ethereum*. Through smart contracts, the supply chain participants are able to track and validate their steps. The smart contracts receive transactions in form of function calls (*e.g.*, `buySeed`, `sellProductToDistributor`, `sellGrainToProcessor`). Some transactions can not be altered, *e.g.* the condition for a sale of grain. Triggers are also in place to monitor events and alert if violations occur.

As this industry makes use of Internet of Things (IoT) enabled items, like containers with GPS, cameras, 4G, or packages equipped with sensors, human interaction with the systems is kept to a minimum. The authors anticipate better availability of verifiable and non-modifiable information. They also raise awareness for the possibility of fraudulent data from stakeholders, as they are anticipated to record proper data.

Bocek et al. [7] focus on a use-case from a Swiss-based start-up that focuses on leveraging BC technology in Supply Chain Tracking (SCT) with various sensors. The main goal is to assure data immutability and public accessibility of temperature records for transparency while reducing operational costs in the pharmaceutical supply chain.

The system is based on the *Ethereum Virtual Machine (EVM)*, where the smart contract is executed. They ensure compliance with the frame conditions (*i.e.*, temperature). A relational database is used to store raw data. A server is in place to communicate between the BC and the front-end users, creating and modifying smart contracts. If data is declared

too sensitive for a public BC or too large, it is stored in a PostgreSQL database. The smart contract is altered so the temperature range can be verified, and is able to store the verification result in the smart contract together with a URL pointing to the raw temperature data and its hash on the server.

As the private Ethereum BC is used in the project of this thesis, the focus also lies on projects with similar prerequisites. They are described in the next paragraph.

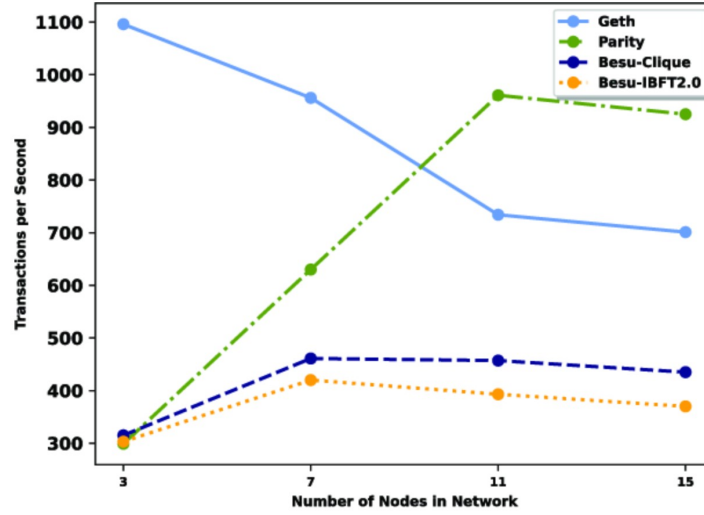


Figure 3.2: Scalability Behavior Testing [37]

Research has gone into evaluating the performance of different settings when deploying a private Ethereum BC in testing environments. The authors of [37], [28] and [20] have shown through numerous experiments how deployment approaches influence the performance of said BC. For example, in paper [37], the authors have shown how the Block Gas Limit (*i.e.*, the maximum amount of gas (or energy) that one is willing to spend on a particular transaction) of different BC clients (*e.g.*, Geth and OpenEthereum (Parity)) impact the transaction performed per second. Another significant variable the authors evaluated is the scalability of the BC clients (see Figure 3.2).

This may become relevant when the CheeseChain project expands and more nodes are included in the network. Setting up a private Ethereum network is relatively easy, as the authors of [20] have shown.

3.2.2 Hyperledger-based Projects

The authors of [45] propose a Supply Chain Tracking for food safety. The system is based on the private and permissioned BC *Hyperledger Fabric*, where permissions are defined by a set of policies and agreed upon by network members when the network is initially configured. The constructed system runs on the Docker Linux platform and allows consumers to retrieve food origin information through a QR code. Their food channel consists of three entities: producer, transporter, and distributor. Each entity has one role: either reader,

writer, or admin, which are identified through public/private keys. The system relies on the implementation of smart contracts to handle the transactions. They are embedded with an “if-this-then-that” code, allowing them to carry out tasks without third-party navigation.

Paper [4] focuses on European food supply chain traceability, and the authors analyzed the implemented system running on the Sawtooth consortium framework of Hyperledger. Sawtooth allows a private, as well as public deployment of the BC framework [4]. It is also possible to have a private release with public access, which was implemented in this case. The architecture of the system is divided into three layers:

1. Physical layer, which contains the products among different organizations within the supply chain.
2. Digital data layer: Holds every data point linked to layer 1, which is useful for traceability.
3. BC layer: Represents the BC platform used for the traceability process.

Through a REST API, layer 2 communicates with the BC (layer 3). The API is also connected to an off-chain Repository, storing credentials of the entities in the system after AES encryption.

Chapter 4

Design and Implementation

This chapter presents the design and the implementation of the solution. The chapter is divided into two sections: Section 4.1 focuses on the design and architecture of the Database-to-Blockchain (DB2BC) tracking system, whilst Section 4.2 will concentrate on the underlying functionalities. Namely, the interaction between the Fromarte Database and the local host, as well as the synchronization to the private and later the public BC.

4.1 Design

As this thesis is a sub-project of the CheeseChain project, the central architecture of the project needs to be introduced to fully understand the outline of this thesis. As illustrated in Figure 4.1, the overall solution grabs information from the Digital Quality Management system (DigitalQM) from Fromarte. After filtering the relevant information, it feeds the data into two BCs: one public and one private. The private BC is semi-decentralized (BC Consortium) since every stakeholder maintains a node. This thesis focuses on the data gathering and the private aspect of the solution.

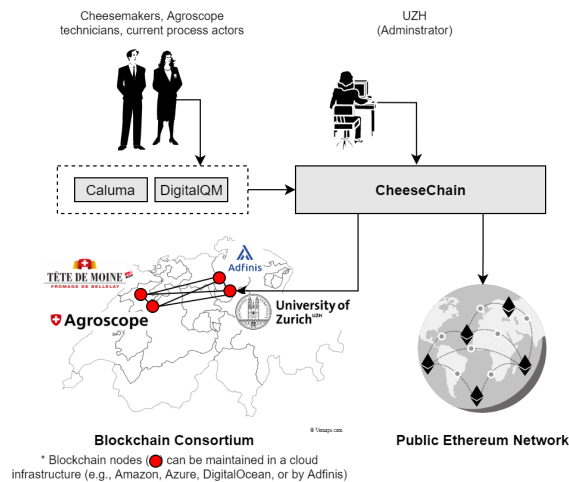


Figure 4.1: Architecture of the overall CheeseChain solution

4.1.1 Data Gathering

At first, the focus lies on gathering the data from the DigitalQM. As seen on the left-hand side of Figure 4.2, whenever a cheese producer fills out a new quality management form, there are two possible states: *finished* or *intermittent*. Either the producer has finished their work and wants to get an OK from the supervisor, or it is intermittent, and needs to be completed at a later time.

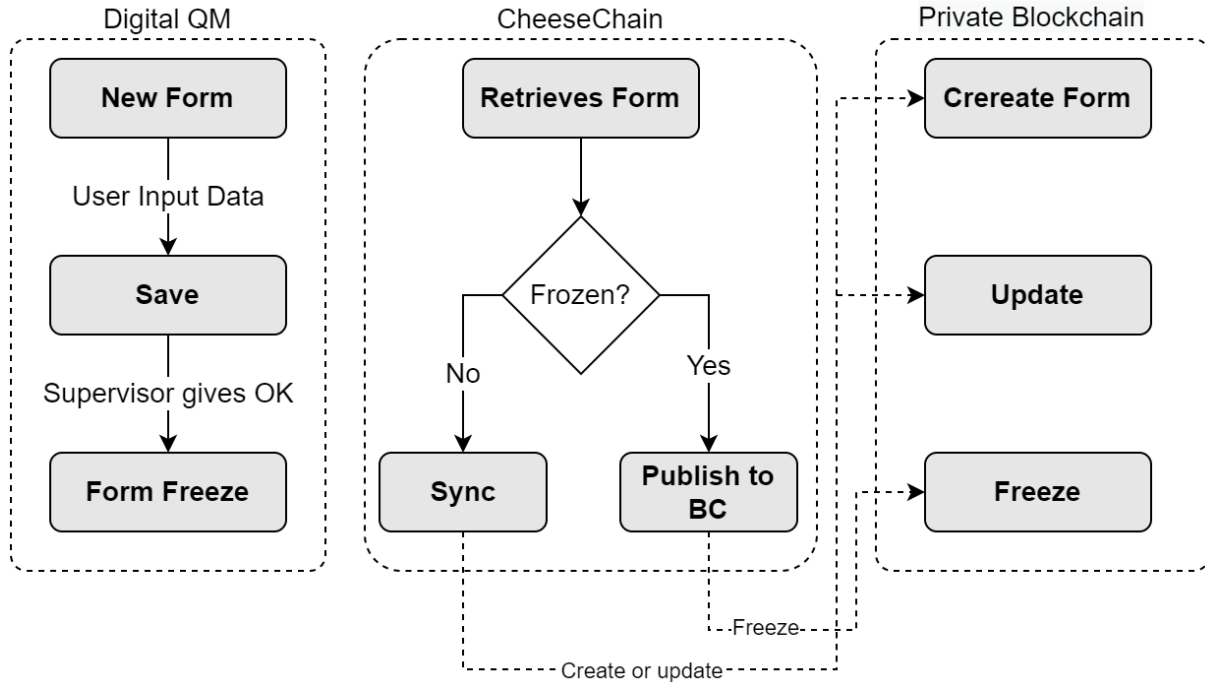


Figure 4.2: Data-workflow

When a supervisor gives their OK, the form is frozen, and cannot be altered anymore. This means a retrieved form through the API of the GraphQL database can have three different statuses: `RUNNING`, `COMPLETED`, or `CANCELED`. At a higher level, the program retrieves the forms, checks if they are frozen (*i.e.*, `COMPLETED`), and if so, publishes them to the private and public BCs. If it is still running (*i.e.*, `RUNNING`), it will be synchronized onto the private BC, to detect possible changes that were made. This process is done in iterative steps to detect all changes.

First, the solution gathers information about every form and saves it. From there on it fetches every form separately to obtain the answers as filled out by the cheese producers. For each retrieved form, the host receives a JSON file, from which the relevant data is extracted, organized, restructured, and saved locally. Finally, it will be pushed onto the appropriate BC. A newly created file with its relevant data is represented as shown in Listing 4.1.


```

1  "RG9jdW11bnQ6MDg2Yzc3ODAtNThlMWQtMjliZmUOMzQ2NWYy": {
2    "createdAt": "2022-07-19T19:27:05.481739+00:00",
3    "createdByUser": "1170",
4    "id": "RG9jdW11bnQ6MDg2Yzc3ODAtNThlMWQtMjliZmUOMzQ2NWYy",
5    "name": "Test Cheese",
6    "lastUpdated": "2022-07-20T12:04:01.782329",
7    "lastModifiedBy": "1170",
8    "status": "RUNNING",
9    "answer": {
10     "833-10-datum": "2022-07-06",
11     "833-10-milchmenge": 7.0,
12     "833-10-temperatur-gelagerte-milch": 8.0,
13     "833-10-lab-lot-nummer": "5"
14   }

```

Listing 4.1: Example JSON pushed onto the BC

The ID of the form is used as a key, whilst the value is a nested JSON containing all important information and answers to certain questions. The relevance of the fields and answers were determined by the supervisors of this thesis.

Whenever a relevant variable is altered, deleted or a new one is given, the solution fetches the answers again, saves the new updated JSON locally and alters the SC state (more in Section 4.1.2). A new saved JSON is always equipped with the UNIX Timestamp [14], for simpler processing later on.

Due to the possibility of numerous open forms, the solution makes use of a functionality called historical answers, which lowers network throughput and preserves the database. It is a less expensive way to check if something was altered than fetching the whole form again. This concept is further elaborated in Section 4.2.

4.1.2 Working with the BC

The interaction between the solution and the BCs is realized through Smart Contracts (SC). The private SC runs on the host and is deployed with the support of two programs. Ganache [13] maintains the platform for the private deployment of Ethereum, whilst REMIX [34] (an IDE) allows the development, deployment, and administration of smart contracts for Ethereum-like BCs. Since tracking of changes is a core functionality of the solution, the SC represents the underlying structure of the DigitalQM.

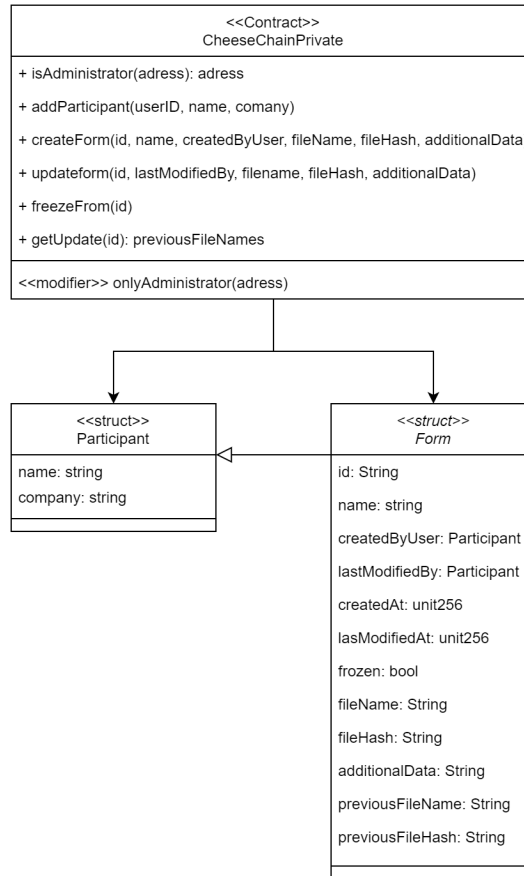


Figure 4.3: Class diagram of the Smart Contract

The SC consists of participants, which is the counterpart to users in the DigitalQM, and forms, which are the forms filled out by cheese producers (see Figure 4.3). Through several functions, the host is able to communicate with the BC through the SC. The *isAdministrator* function checks whether a proposed transaction comes from a trustworthy address. It is embedded into several other functions to make sure no unauthorized address alters anything. *CreateForm* creates a new form instance, while *updateForm* and *freezeForm* both alter the state of file instances. *E.g.*, *freezeFrom* alters the state of the bool *frozen* from false to true, when called.

Each new filled out form in the DigitalQM creates a new form instance in the BC. As soon as a form is altered or updated, the instance will be updated with the current data and appointed a string with the name of the past file, forming a unidirectional linked list, as illustrated in Figure 4.4.

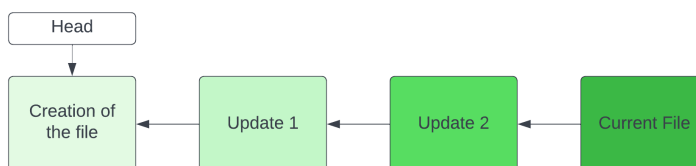


Figure 4.4: Illustration of linkage between a file and its past versions on the BC

To ensure data integrity of past saved forms, the hashes of the old files are saved in a similar manner as the filenames. When fetching outdated files from the host, the file can be hashed again and compared to the hash on the BC, to be sure they are in the same state as when they were saved.

4.1.3 On- off-chain data storage

Storing the extracted data in a secure fashion is a crucial point in the design of the solution. Since some of the data is sensitive, storing it on the BC is not advisable. Hence, the design of the solution provides on- and off-chain storage. As stated in Section 4.1.2, the hashes of the current, as well as previous files are saved in a string on the BC, to ensure data integrity. Since the names of the files are stored in the same way (see Figure 4.5), a simple API can fetch past files, since the filenames are equipped with the Unix timestamp to identify the corresponding file. After checking whether they still have the same hash, the API can return the gathered file.

RG9jdW11bnQ6MDg2Yzc3ODAtNDZU0MzQ2NWYy-1658318494723259.json

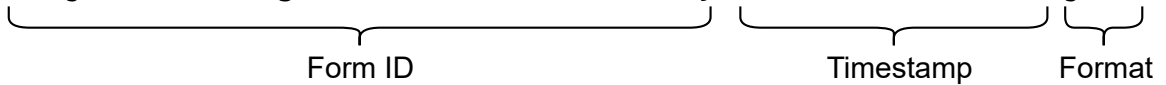


Figure 4.5: File Format Stored in the Server

This method ensures that the BC only contains necessary data, and no sensitive information will be exposed in the private or the public BC.

4.2 Implementation

The thesis implements the presented design in Figure 4.1 and Figure 4.2 as a command line tool using Python, Solidity, and the GraphQL query language. The code and instructions are available at [15]. The solution of this thesis runs in an infinite loop and can be interrupted through the command line. The implementation is split into three main parts: **Data Gathering**, **Data Parsing** and **Data Synchronization** onto the BC.

4.2.1 Data Gathering

In the beginning, all the forms need to be fetched from the DigitalQM database. This is accomplished through an API call to the GraphQL endpoint. Through the `AIOHTTPTransport` library, which allows the sending of GraphQL queries using the HTTP protocol, the form can be fetched. For the execution, a query and a valid authorization token (transported inside the header) are needed. To obtain a valid OAuth 2.0 authorization token (short-lived), a refresh token (long-lived) is needed, which initially needs to be copied from

the fromarte web page itself. The solution then is self-sufficient, meaning it will always save the newest and still valid refresh token in the *Config.json* file. The Python function in Listing 4.2 is used to fetch a document from the database.

```

1  def getDocument(self, docID, query, dateTime=False):
2      transport = AIOHTTPTransport(url="https://beta.qs.fromarte.ch/graphql/",
3                                  headers={"authorization": "Bearer " + self.getAuthToekn()})
4      client = Client(transport=transport)
5      # Provide a GraphQL query with the docID
6      query = gql(query)
7      params = {"id": docID}
8      # DateTime may be relevant when fetching historical answers
9      if dateTime:
10         params['dateTime'] = self.dateTimeIso
11     response = client.execute(query, variable_values=params)
12     return response

```

Listing 4.2: Fetching of forms

As authorization is needed to fetch from the DigitalQM database, the function in Listing 4.2 needs to include said authorization token (e.g., 9d3194ba805045afbe4e1ab24a1b4079). After logging in, the host receives both an authorization and a refresh token. The latter is needed to fetch a new authorization/refresh token pair. This process needs to be completed in iterative steps as the authorization token will be invalid after 300 seconds.

To query all the forms and their relevant fields, the following Graph Query Language (GQL) query is passed into the `getDocument` function. GQL allows setting filters to their queries, so the returned JSON is already usable. In the case of Listing 4.3 ordered in a descending fashion, oriented at the `createdAt` variable. What is more, only the files with the status `RUNNING` or `COMPLETED` will be returned, as files that were `CANCELED` or `SUSPENDED` are not relevant for the solution.

```

1  allCases(orderBy: CREATED_AT_DESC, status: [RUNNING, COMPLETED]) {
2      edges {
3          node {
4              createdAt
5              createdByUser
6              document {
7                  id
8                  form {
9                      name
10                 }
11             }
12             status
13         }
14     }

```

Listing 4.3: Query for all forms

The response is a single nested JSON file, containing nodes of all forms saved in the Fromarte database. It will then be passed onto the next function, extracting all the relevant information and rearranging it into a new organized JSON.

```

1  def getRelevantInfoAllWorkingItems(self, json):
2
3      final = {}
4      if type(json) == dict:
5          for k, v in json.items():

```

```

6         if type(v) == dict or type(v) == list:
7             final = final | self.getRelevantInfoAllWorkingItems(v)
8         if k in self.searchFilterAllWorkingItems:
9             final[k] = v
10    elif type(json) == list:
11        for element in json:
12            final = final | self.getRelevantInfoAllWorkingItems(element)
13    return final

```

Listing 4.4: Extracting relevant variables from the response JSON

Listing 4.4 shows the function designed to extract predefined variables stored in the variable (`searchFilterAllWorkingItems`) inside the config file, containing *i.e.*, `created-ByUser`, `createdAt`, `id`, `name` and `status`. As the received JSON is nested, the most applicable approach is a recursive function. The function calls itself whenever it is faced with a new dictionary or list. The recursion makes the solution more agile and easier to extend. The drawback is that the runtime complexity increases and the computation gets more expensive. In the case of this function, however, it can be viewed as insignificant. It should also be noted that if there are two variables in the JSON with the same name, the function will always take the latter since the first one would be overwritten.

4.2.2 Data Parsing

After the IDs of all forms have been fetched, the corresponding answers need to be fetched and stored. The script runs through the stored IDs and retrieves the answer file separately for each form. This is achieved through a complex GQL query, which can be found in (Appendix A).

Similarly to the function in Listing 4.4 the `getRelevantInfoFromJsonAnswers` function in Listing 4.5 extracts certain predefined variables. Due to the complexity of the returned answers-JSON, the approach to the recursive structure needs to be adapted. The recursion implies again that the function is able to handle changes in the received JSON, as long as the structure of the nodes (see Listing 4.6) stays the same. The extension of the set of relevant variables (`search_words` in Listing 4.5) is also guaranteed, thanks to the dynamicity of the function.

```

1     def getRelevantInfoFromJsonAnswers(self, jsonname, search_words=None):
2         # take the predefined searchwords, if the variable is not defined.
3         if search_words is None:
4             search_words = self.searchFilterMilkRelated
5         final = {}
6         # check if a name of a file is given, or a whole file is passed in the function
7         if type(jsonname) == str:
8             with open(jsonname, encoding='utf-8') as f:
9                 loaded = json.load(f)
10        else:
11            loaded = jsonname
12
13        # check all answer nodes for relevant variables
14        if type(loaded) == dict:
15            for k, v in loaded.items():
16                if type(v) == dict or type(v) == list:
17                    final = final | self.getRelevantInfoFromJsonAnswers(v, search_words)
18                if v in search_words:
19                    final[v] = "tbt"
20                if "tbt" in final.values():

```

```

21         if "Value" in k:
22             final.update({ke: v for ke, val in final.items() if type(v) in [
23                 str, float, int]})
24             elif type(loaded) == list:
25                 for element in loaded:
26                     final = final | self.getRelevantInfoFromJsonAnswers(element, search_words
27             )
28
29     return final

```

Listing 4.5: Extracting relevant answers given by CheeseProducers

If a file's name is passed into the function, it loads the data with the help of the JSON package. Otherwise, a JSON is given as an argument, so it does not need to load anything from the repository. On line 14 in Listing 4.5, the function first checks if the loaded JSON is of type dictionary or else of type list (on line 23 in Listing 4.5). If it is a list, the function loops through the list and calls itself again for each item. The result received gets appended to the current `final` dictionary. In the case of a dictionary, the function loops through the key:value pairs and checks (on line 14 in Listing 4.5) whether the value is of type list or dictionary. If true, similar to line 23 in Listing 4.4, the function calls itself and appends the result to the `final` dictionary variable.

In the second if-conditional on line 16, the function checks whether the question is of relevance. Since the given answer value is stored one layer outside of the question name (See line 7 for the name and line 10 for the value in Listing 4.6), the `final` dictionary is updated with the question as key and a placeholder “tbt” as value.

The last conditional inside the `for` loop is set to extract the corresponding value to the extracted question. Since the question was saved with “tbt” as value, the statement on line 18 in Listing 4.5 checks whether there is even an answer value to extract, by looking for “tbt” in the values of the `final` dictionary. If it was found, a second nested conditional is reviewed to see if the string “Value” is inside the key, as answer values have “stringValue”, “floatValue” or “integerValue” as a key.

```

1  ...
2  {
3      "node": {
4          "id": "U3RyaW5nQW5zd2NzY1YFhLNC05NGQwLTNhOTcxY2NmYTBlYg==",
5          "question": {
6              "id": "VGV4dFF1ZXN09u0jMC1sYWItbG90LW51bW1lcg==",
7              "slug": "833-10-lab-lot-nummer",
8              "__typename": "TextQuestion"
9          },
10         "stringValue": "22",
11         "__typename": "StringAnswer"
12     },
13     "__typename": "AnswerEdge"
14 }
15 ...

```

Listing 4.6: Example of an answer node

Similar to the Function in Listing 4.4, if there are two variables with the same name in the answer JSON, the function will always take the latter.

4.2.3 Data Synchronization

After fetching all existing forms, the main focus lies on updating the existing data and checking for new forms. As already stated, the database of the DigitalQM saves information on answer changes in the forms. Namely, the user ID, time, and whether a new answer was given (indicated by a +) or an answer was altered or deleted (displayed both times through a ~). For this, the solution iterates through all existing files and fetches the historical answers. In the same API request, the status (*e.g.*, RUNNING) of the file is also fetched (see Listing 4.8), to see whether the form was frozen.

```

1 def updateFiles(self):
2     ...
3     for historical_answer in returnedJson["historicalAnswers"][ 'edges' ]:
4         question = historical_answer["node"]["question"]["slug"]
5         hist_date = historical_answer["node"]["historyDate"]
6         if historical_answer["node"]["historyType"] == "~" and question in self.
searchFilter and newest_history_date != hist_date[: (hist_date.index("+"))]:
7             # something was altered or deleted and is new
8             new_answers.append(question)
9             if dateutil.parser.parse(newest_history_date) < dateutil.parser.parse(
10                 hist_date[: (hist_date.index("+"))]):
11                 newest_history_date = hist_date[: (hist_date.index("+"))]
12             # save the Username
13             modifiedByWho = historical_answer["node"]["historyUserId"]
14             elif question in self.searchFilter and question not in node[1][ 'answer' ]:
15                 # check if a new relevant answer was given
16                 new_answers.append(question)
17                 if dateutil.parser.parse(newest_history_date) < dateutil.parser.parse(
hist_date[: (hist_date.index("+"))]):
18                     newest_history_date = hist_date[: (hist_date.index("+"))]
19     ...

```

Listing 4.7: Checking for new answers

The `updateFiles` function runs through each historical answer node (*e.g.*, Listing 4.8), and evaluates whether the answer to the question needs to be fetched again. There are three conditions that need to be met (see Listing 4.7 Line 6):

1. The “historyType” (Line 7 in Listing 4.8) needs to be a ~ or a +.
2. The question needs to be relevant, meaning they need to be in the list of the question to be extracted.
3. The “historyDate” cannot be equal to the variable “lastUpdated” (see Line 5 on Listing 4.8) timestamp otherwise the same changes would be fetched again.

If all the listed conditions are met, the function appends the question to the `new_answers` list. As it could be that a new answer was given to a question that was not answered before, the script needs to check this case. This is achieved on line 14 in Listing 4.7. The evaluation, whether the question is of relevance and has never been picked up before, takes place in the `elif` statement.

In both cases, the `historyDate` is updated, if it is newer than the last time the form was checked (*e.g.*, Line 17 in Listing 4.7).

```

1  ...
2  {
3      "node": {
4          "createdAt": "2022-07-19T14:07:05.859337+00:00",
5          "historyDate": "2022-07-19T18:08:27.480661+00:00",
6          "historyUserId": "1170",
7          "historyType": "~",
8          "createdByUser": "1170",
9          "question": {
10             "slug": "833-10-temperatur-einlaben-massnahme",
11             "label": "Massnahme zu Temperatur"
12         }
13     }
14 }
15 "node": {
16     "case": {
17         "status": "RUNNING"
18     }
19 }
20 ...

```

Listing 4.8: Example of a historical answer node and its status

If a change was picked up, the next step is to fetch the provided answers. Similarly to when first fetching and extracting the variables, the solution makes use of the `getRelevantInfoFromJsonAnswers` function. In this instance, the list `new_answers` is passed as the `search_words` argument. Thus the extracting function knows which variables need to be extracted.

```

1  def updateFiles(self):
2      ...
3  for question, newVal in new_fetched_answers.items():
4      try:
5          if node[1]["answer"][question] or node[1]["answer"][question] == None:
6              if node[1]["answer"][question] != newVal:
7                  # the new value was really new
8                  data[node[0]]["answer"][question] = newVal
9                  total_changes += 1
10                 changes_to_current_file += 1
11                 if node[0] not in ids_of_updated_files:
12                     ids_of_updated_files.append(node[0])
13     except:
14         # Catch if the question does not exist
15         data[node[0]]["answer"][question] = newVal
16         total_changes += 1
17         changes_to_current_file += 1
18         if node[0] not in ids_of_updated_files:
19             ids_of_updated_files.append(node[0])
20
21 # check if a given value was deleted
22 for quest in new_answers:
23     if quest not in new_fetched_answers and quest in data[node[0]]["answer"].keys():
24         data[node[0]]["answer"].update({quest: None})
25         total_changes += 1
26         changes_to_current_file += 1
27         if node[0] not in ids_of_updated_files:
28             ids_of_updated_files.append(node[0])
29 ...

```

Listing 4.9: Examine the new answers

While looping through the new fetched answers, the function needs to evaluate several cases. First, the function tries to find the current question-answer pair. As it is possible that the if-statement will not find anything, the condition is inside a `try` statement to catch the exception of the question-answer pair not being present yet. The if statement on Line 5 in Listing 4.9 needs to check whether the question-answer pair is present or if the key is equal to `None`, as it will be set to `None` when an already given answer is deleted.

Before altering the current value to the new value, the function checks if it really is new, since an unnecessary update could be made, which could trigger an irrelevant transaction on the BC. Whenever the value is new, it gets altered alongside the increase of some counters, which are relevant later, to know if changes were made. Lastly, the ID of the altered form is appended to a list, which will be returned at the end of the function.

As stated above, the `except` catches the exception if the question-answer pair is not present in the corresponding *answers* sub-JSON (see Line 9 - 14 in Listing 4.1). In this case, the exception creates a new entry inside the *answer* sub-JSON.

The last possibility to alter an answer is to delete an existing one. This case is covered in the `for` loop on line 22 in Listing 4.9. Contrary to the loop on Line 3 in Listing 4.9, this `for` loop iterates through the `new_answers` list, in which all the altered questions are saved from the historical answers. Since there is no answer to a question, the database will not return anything when fetching the answers. Therefore, to check if an answer was deleted, the question to the answer needs to be in the `new_answers` list but not inside the `new_fetched_answers` list. On line 23 in Listing 4.9, the `if` condition checks exactly this. If it is evaluated as true, the answer (value) to the question is updated and set to `None`. Like in every other conditional, two counters are increased and the ID is appended to the `ids_of_updated_files` list.

The two counters `changes_to_current_file` and `total_changes` let the program know when to save a new file.

```

1  def updateFiles(self):
2  ...
3      if changes_to_current_file != 0:
4          # update the lastUpdated time
5          data[node[0]]["lastUpdated"] = newest_history_date
6          data[node[0]]["lastModifiedBy"] = modifiedByWho
7          local_name = node[0] + r"-" + str(time.time())
8          local_names.append(local_name)
9          os.chdir("BackupFiles")
10 ...
11 ...
12     if total_changes != 0:
13         # Update nameNewestBackupFile variable
14         self.nameNewestBackupFile = "BackUp" + str(time.time()) + ".json"
15         with open(new_name, "w", encoding='utf-8') as f:
16             json.dump(data, f, indent=2)
17     return ids_of_updated_files, ids_to_freeze, local_names

```

Listing 4.10: Saving the new files

If `changes_to_current_file` (see Line 3 in Listing 4.10) is positive, the code inside the `if` statement will be triggered. The *lastUpdated* (see Line 6 in Listing 4.1) value will be set to the most current update time from the historical answers (see Line 5 in Listing 4.8). The username of the user who did the changes will be saved in the *lastModifiedBy* value.

The `local_name` will be saved in the `local_names` list, which will also be returned at the end. The `local_names` list is needed to have a consistent naming of the files on the host and the BC, as they are equipped with the Unix timestamp.

After saving the files separately (Lines 3 - 9 in Listing 4.10) in the *BackupFiles* folder, the *BackUp*, containing all RUNNING files, is updated and saved. Finally, all IDs of the updated files (`ids_of_updated_files`), the IDs of the frozen files (`ids_to_freeze`), and the local names (*e.g.*, RG9jdW1lbnQ6MDcWQligZmU0MzQ2NlYy-1658666712.3016922, in the `listlocal_names`) are returned.

4.2.4 Interaction with the private BC

Below, the interaction between the script and the private Blockchain is described. The sections will be split up to represent the lifecycle of forms. First the **creation**, then the **synchronization** and lastly the **freezing** of the forms.

Creating an instance on the SC

As stated in Section 4.1, the platform Ganache [13] is used for the private deployment of Ethereum. With the help of Remix [34] for the deployment of the Smart Contract, the BC is set up. Web3 [32] is a Python package used for interacting with Ethereum running on Ganache.

The lifecycle of an instance on the BC starts with the creation of a new form in the DigitalQM. After the data extraction (described in the Section 4.2.1), a new transaction is triggered (see Line 14 - 22 in Listing 4.11).

```

1  class CommunicateToSmartContract:
2      def __init__(self):
3          ...
4          self.contract_instance = self.w3.eth.contract(address=self.ScAddress, abi=self.
          abi)
5          ...
6      def createHash(self, jsontohash):
7          """
8          Hashes a JSON with the Sha 256 hash.
9          :param jsontohash: The file (dictionary) that needs to be hashed.
10         :return:
11         """
12         return hashlib.sha256(json.dumps(jsontohash).encode('utf8')).hexdigest()
13
14     def createNewFormSmartContract(self, id, data, name_of_file):
15         """
16         Creates a new instance on the Blockchain, with the given parameters.
17         :param id: ID of the form
18         :param data: The file itself
19         :param name_of_file: Name of the local file. "ID-Unix timestamp"
20         """
21         self.contract_instance.functions.createForm(id, data["name"], 1391, name_of_file,
22             self.createHash(data), "").transact({'from': self.__mySCAdress})

```

Listing 4.11: Transacting onto the Blockchain

A contract instance in the script (see Line 4 in Listing 4.11) consists of the SC address, the address of the sender (meaning the one who created it), and the Application Binary Interface (ABI). The ABI is a standard way to interact with contracts in the Ethereum ecosystem. It represents the structure of the SC in the JSON format, to let the script know how the contract is built.

As the class diagram states in Figure 4.3, the function `createForm` on SC is composed of several variables from the form:

- *id* (e.g., RG9jdW1lbnQ6MDcWQligZmU0MzQ2NWYy)
- *name* (e.g., Fabrikation Tête de Moine) the name of the production
- *createdByUser* (e.g., 1391) user ID from the DigitalQM
- *name_of_file* (e.g., RG9j...Q2NWYy-1658666299.8296485.json) the local name of the file
- *fileHash* (e.g., 4a83ba56d0a735ef97befc488bfcd3ffe4974c523a4ba266f19a88a120) which represents the hashed file. See Line 12 in Listing 4.11
- *additionalData* a space for additional data to save on the BC

The administrator's address from the SC must be included in the transaction for the transaction to be authorized. The authorization address is created when the SC is initially deployed.

Synchronization

After updating the current files (see Section 4.2.3) and checking for new ones, the host knows what happened between the last iteration intervals. Next, the new data needs to be synchronized onto the private Blockchain.

```

1 class CommunicateToSmartContract:
2     ...
3     def updateFormOnSmartContract(self, id, BackUpFileName, name_of_file):
4         """
5         Updates a form instance on the Blockchain with the given parameters.
6         :param id: ID of the form
7         :param BackUpFileName: Name of the BackUpFile.
8         :param name_of_file: Name of the local file. "ID-Unix timestamp"
9         """
10        self.contract_instance.functions.updateForm(id, 1391,
11                                                    name_of_file, self.createHash(data), "").transact({'from': self
12                                                    .__mySCAdress})

```

Listing 4.12: Updating the state of an instance on the BC

The process of updating a form is similar to creating one. The following variables stay the same as the listed variables in **Creating an instance on the SC**:

- *id*

- *name_of_file*, with the new Unix timestamp
- *fileHash*, with the new hashed file

The *lastModifiedBy* variable is newly introduced and represents the user id of the user who performed the modification to the form on the DigitalQM.

Freeze form

The freezing of the forms is rather uncomplicated. A simple transaction is triggered when a form is frozen (*i.e.*, status: COMPLETED).

```
1 class CommunicateToSmartContract:
2     ...
3     def freezeForm(self, id):
4         """
5         Freezes a form on the Blockchain
6         :param id: ID of the form that needs to be frozen.
7         """
8         self.contract_instance.functions.freezeForm(id).
9             transact({'from': self.__mySCAddress})
```

Listing 4.13: Freeze a form on the BC

The `freezeForm` takes the *id* of a form as an argument and `transacts` with the address of the administrator (see Line 8 - 9 in Listing 4.13). The SC instance's bool *frozen* will be changed to `true`, and the form can not be altered afterward.

Chapter 5

Evaluation

This chapter evaluates the design and implementation in the context of the CheeseChain project. While Chapter 4 focused on technical aspects (*e.g.*, design and implementation), this chapter discusses the solution’s performance in different settings. The goal is to determine how the solution behaves in certain circumstances (*e.g.*, the influence the number of forms, the fetching, parsing, and BC interaction time has on the solution’s overall performance).

5.1 Relation to the CheeseChain Project

The following questions will be answered in this section. The first is “*Which challenges does the solution face?*”, while the second is “*Where can it be integrated into the overall project?*” To recap the purpose of the CheeseChain according to [12]:

The CheeseChain project targets the development and implementation of a platform to improve transparency and trust along the Tête-de-Moine value chain.

With the implemented solution, which utilizes the advantages of the BC technology, this POC provides valuable insights into how the goals of the CheeseChain project may be achieved. The solution covers the private part of the CheeseChain project. Through a private BC, the solution tracks changes in an immutable fashion which can be viewed as a favorable design to bring more *transparency and trust* along the value chains of cheese dairies. The solution tackles the mentioned challenges in the following way: The hybrid on- and off-chain data storage pairs the best of both worlds. Off-chain data storage tends to be cheaper and easily extendable, while the BC is immutable, decentralized, and incorruptible. Paired through an API it is a competitive solution for storing data in a secure way.

Not only is food safety guaranteed for the producers and vendors, but the end consumers also have additional value with such a solution in place. As forgery has become harder to

spot, even with proof of origin logos, such a solution with ensured correctness is a valuable asset.

5.2 Performance Testing

To assess the performance of the solution, different aspects were taken into consideration. Most notable is the network. As the solution fetches several times from the DigitalQM endpoint, the accuracy of the measured performance may diverge when remeasuring due to network limitations or database limits. The tests were performed using an HP Spectre with an Intel Core i7-1165G7 quad-core CPU running at 2.8 GHz/4.7 GHz, 16 GB of RAM, and 1 TB SSD connected to LAN so that the connection with the DigitalQM is as stable as possible.

5.2.1 Run-time

The run-time of functions or certain parts of the code can be measured quite easily with the package time [33]. This may look as such:

```

1 def doSomething(self):
2     ...
3     st = time.time()
4     # Code to be measured
5     for id, val in final.items():
6         c.createNewFormSmartContract(id, val, [item for item in local_names if item.
           startswith(id)][0])
7     et = time.time()
8     print('Execution time:', et - st, 'seconds')
9     ...

```

Listing 5.1: Measuring the rin time of a function

The floating point number `time.time()` saves the current epoch (since January 1. 1970) Unix time in the variable `st` (*i.e.*, starting time) or `et` (*i.e.*, end time). To determine how long it took, the difference is measured by subtracting the end time from the starting time (*i.e.*, `st - et`).

Different aspects of relevance (*e.g.*, functions or loops) were examined to test the solution's performance. Below are the main tests performed for the evaluation:

1. Time until the whole database of the DigitalQM is mirrored onto the host (fetching see Section 4.2.1, backup files see Section 4.2.2) and on to the BC (see Section 4.2.4) and the forms are updated and frozen (see Section 4.2.4).
2. Time until the whole database of the DigitalQM is mirrored onto the host (fetching see Section 4.2.1, backup files see Section 4.2.2) and on to the BC (see Section 4.2.4).
3. Time until the whole database of the DigitalQM is mirrored onto the host and only the host (see Section 4.2.1 and Section 4.2.2)

4. Time until every form is mirrored onto the BC (see Section 4.2.4), when running for the first time.
5. Time until every answer from is fetched (see Section 4.2.2)
6. Time until the relevant fields are extracted from each form *i.e.*, parsed (see Section 4.2.2)
7. Time until every from is frozen (*i.e.*, COMPLETED) see Section 4.2.4 and Section 4.2.3
8. Time until every form is updated, when there is nothing to update (see Section 4.2.3), and if there is one form to update.

Currently, there are 27 forms on the DigitalQm (as of 05.08.2022), consisting of 8 open (*i.e.*, RUNNING) and 19 frozen (*i.e.*, COMPLETED) forms. The starting times are taken before a function call, or right before starting a loop. In contrast, the ending times were taken after the function call, before the last (**return**) statement, or directly after a loop. In Table 5.2.1, the result of the measured times are presented and compared. The results in seconds are set in relation to the whole iteration time, including the fetching of all forms, fetching of the answer files, parsing of the data, and synchronization onto the BC.

Table 5.1: Performance Test Results

Test	Description	Run-time [s]	Percentage*
1	Database onto host and BC, update and freeze	88.474	100.00%
2	Database onto host and BC	73.983	83.62%
3	Database onto host	12.384	13.99%
4	Mirroring onto the BC	60.533	68.41%
5	Fetching	12.250	13.84%
6	Parsing data	0.115	0.12%
7	Freeze every form	9.746	11.01%
8	Update every form	4.081	4.61%

* In relation to Test #1

Not every line of code (*e.g.*, directory changes, opening, closing, and saving of JSONs) is covered in the performance tests (*i.e.*, 1 - 8 in the Table 5.2.1), so the percentages need to be evaluated with caution. However, they still provide a valuable insight into the duration of the different steps of the solution. The sum of all the percentages will not equal 100, since some tests cover functionalities which are also covered by other tests. The run-times will most likely give different results each time they are measured, due to network instabilities, lags, and database workload.

Table 5.2.1 shows that the whole first iteration takes 88.474 seconds (*i.e.*, fetching, parsing, saving on the host and BC, freezing, and updating once), whereof 70.279 seconds are used for the BC interaction (*i.e.*, freezing or creating instances). Data parsing, fetching, and updating (*i.e.*, re-fetching the RUNNING files) combined make up 16.5 seconds or 18.5% of the first iteration.

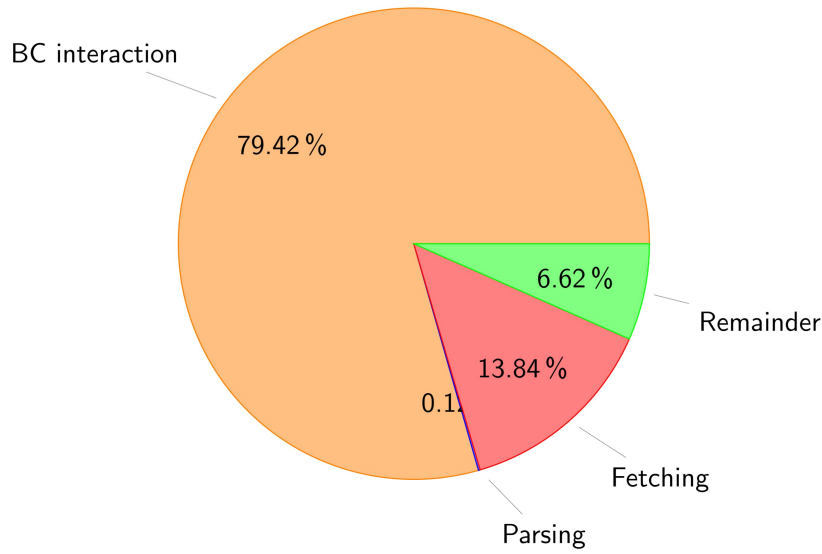


Figure 5.1: Illustration of time usage for BC interaction, fetching, and parsing

Figure 5.1 offers an illustrative visualization of how the time of the solution is divided. The BC interaction with the private BC takes almost 80% of the time (*i.e.*, freezing and creating instances). Fetching each answer-file and re-fetching the RUNNING ones take 13.8%, while parsing is quite fast with 0.12%. The remainder, *i.e.*, opening, closing, saving JSONs, and directory changes take up 6.6% of the time.

It is important to note that the interaction time with the BC strongly depends on how many COMPLETE forms are on the DigitalQM, since the solution first creates the instance on the BC and freezes it afterward (*i.e.*, two calls to the BC).

5.2.2 Scalability

As scalability is crucial for productive environments, it needs to be analyzed in this POC. The bottlenecks of the private BC were already discussed in Section 5.2.1. Each timing test listed (in Table 5.2.1) was also performed with one to ten items in the DigitalQM to comprehend the solution's scalability. None were frozen, meaning there were fewer BC interactions than in the tests in Table 5.2.1.

Figure 5.2 shows how time stands in relation to the number of forms in the DigitalQM. The plot indicates a linear trend with increasing forms. This can be explained through the BC interaction. Since the creation of instances always takes the same amount of time. If the freezing time from test 1 were subtracted and provided with an error margin, test 1 would be in the predictions margins of Figure 5.2.

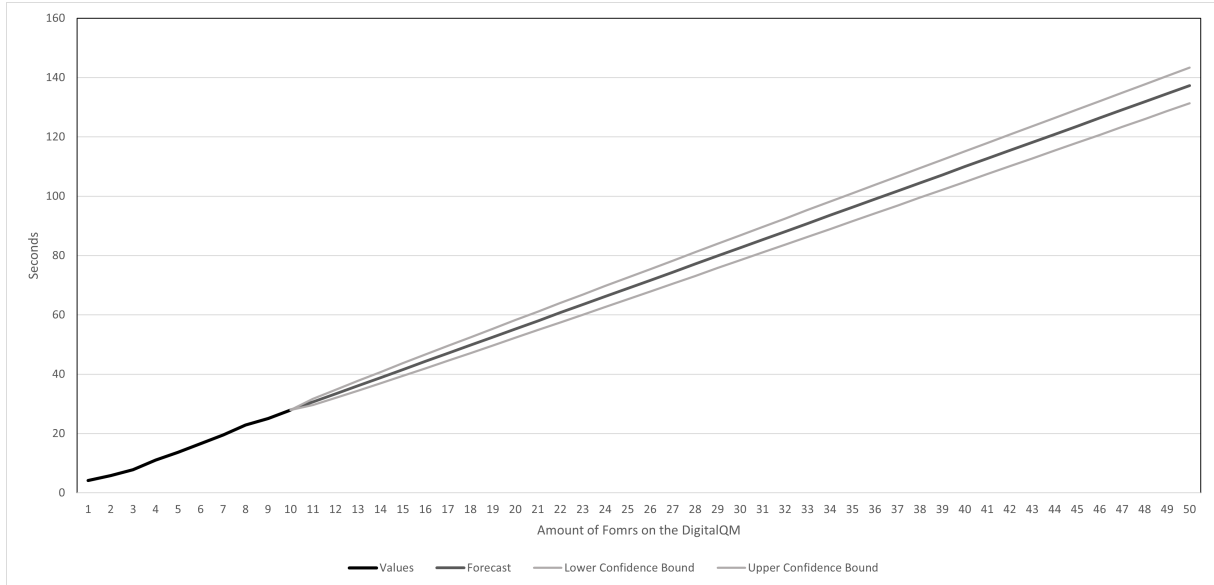


Figure 5.2: Scalability of the solution

5.3 Limitations

In this section, the focus lies on the resistance of the solution and its limitations. There are two main parts to focus on when analyzing the resistance of the solution; the *code* itself, and the GraphQL *queries*.

5.3.1 Code limitaitons

The implemented solution itself is more or less self-sufficient, except for the libraries. As stated in Section 4.2.2 the extracting functions are implemented in a recursive fashion, which brings the benefit of easy extensibility. The downside of the recursive implementation is that it is not possible to detect two or more variables with the same name, the last encountered match will always be used. The `search_words` *i.e.*, the words which will be backed up, can comfortably be changed or extended without any problems. For convenience, they are saved in a *Confog.json* file, alongside all the GraphQL queries, the SC Application Binary Interface (ABI), SC addresses, and the refresh token for the DigitalQM.

5.3.2 Query limitations

The queries presented in Section 4.2.2 and Appendix A may also be modified or altered since the code will be able to handle changes. If namings change in the fromarte GraphQL database, the queries will fail. The queries are also saved in the *Config.json* file, so the code does not need to be touched when making modifications.

Chapter 6

Summary, Conclusion, and Future Work

The ever-advancing globalization and digitalization led to new markets and progressive opportunities. In a fast-evolving world, the structures building the support system also need to adapt. Delays, exploitation, and forgery bring the need for secure systems. Even though the BC technology needed to evolve to the Blockchain 2.0 era in order to open up the platform for more than cryptocurrencies, it can be viewed as revolutionary for several industries (*e.g.*, pharma and food). The objective of this thesis was to design and implement a solution for the CheeseChain project to achieve more transparency and trust along the value chain of Fromarte. the aim of the solution is to gather data from their DigitalQM GraphQL database, extract relevant data, organize it in a clear way, and store it in a folder and on local BC.

A conceptual introduction was given to (private-) blockchains and supply chains to be able to develop and comprehend the project at hand. For BCs, features and functionality, transaction structures, consensus mechanism designs, and two private BCs (*i.e.*, Ethereum and Hyperledger) were introduced. Existing re-engineering supply chain methods, private BC industry projects, and hybrid (*i.e.*, on- and off-chain) database designs were analyzed and surveyed through related work. With the insights gathered by analysing the Background and related work, the design and implementation of the solution were developed for this thesis.

The data gathering, parsing, and BC interacting tool was created as a software solution. It interacts directly with the GraphQL endpoint of the DigitalQM database, where the files get fetched. On the host, the files get saved locally and then pushed onto the local BC. Through performance testing, it was determined that the BC interaction takes up most of the run-time of the solution. Since the authentication token is valid for 300 seconds, the solution as it is could not manage a large (150+ forms) database, as the BC interaction would take more than said time.

6.1 Conclusion

In conclusion, the developed solution sets a future-oriented framework for a more security-driven data management system (DigitalQM) for Fromarte. Paired with an API control-

ling the interaction between the host and the BC, the solution provides additional value to the current system. The saved and immutable hashes of the backup files stored on the BC deliver the required security to cheese diaries and end consumers. Through a comparison of the stored hash of a file and a newly generated one, the integrity of the file can be guaranteed.

As the solution was not tested in an active environment with real-world data, it can only be anticipated how it would behave. Through the evaluation in Chapter 5, the following assumption can be made (as long as the database is built up the same way as the one of Fromarte): The data gathering (see Section 4.2.1) and data parsing (see Section 4.2.2) will act as anticipated, without any delays. As uncovered in Section 5.2.1, the bottleneck of the solution is the interaction with the private BC, especially when fetching the whole database for the first time.

6.2 Future Work

Future work may include a more precise GraphQL query structure, as they were not designed exactly for the interaction they are used for in the solution. This means that, unfortunately, redundant data is fetched as well. To bring the solution closer to a pragmatic, commercial-ready deployment, a private Ethereum environment with several nodes in the network should be set up, to test how the solution performs in said setting. The solution should also be adapted to be compatible with a database containing more than 150 forms.

Concerning releasing the solution in a commercial environment, the safety of the saved backup files needs to be taken into consideration, even though the integrity is guaranteed through the BC. For aesthetic reasons, a user interface could be designed and implemented for customers and cheese producers, to track the products and check their authenticity.

Bibliography

- [1] Antonopoulos, A.M. and Wood, G. and Wood, G. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018.
- [2] Antonopoulos, Andreas M. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1st edition, 2014.
- [3] Antonopoulos, Andreas M. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1st edition, 2014.
- [4] Baralla, Gavina and Pinna, Andrea and Corrias, Giacomo. Ensure traceability in european food supply chain by using a blockchain system. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 40–47, 2019.
- [5] Belotti, Marianna and Božić, Nikola and Pujolle, Guy and Secci, Stefano. A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys Tutorials*, 21(4):3796–3838, 2019.
- [6] Bhutta, Muhammad Nasir Mumtaz and Ahmad, Muneer. Secure identification, traceability and real-time tracking of agricultural food supply during transportation using internet of things. *IEEE Access*, 9:65660–65675, 2021.
- [7] Bocek, Thomas and Rodrigues, Bruno B. and Strasser, Tim and Stiller, Burkhard. Blockchains everywhere - a use-case of blockchains in the pharma supply-chain. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 772–777, 2017.
- [8] Buch, Hitarshi. Is Hybrid Blockchain the Future?, 2021. <https://www.wipro.com/blockchain/is-hybrid-blockchain-the-future/>, Last visit March 2, 2022.
- [9] Chang, Shuchih Ernest and Chen, Yi-Chian and Lu, Ming-Fang. Supply chain re-engineering using blockchain technology: A case of smart contract based tracking process. *Technological Forecasting and Social Change*, 144:1–11, 2019.
- [10] Christidis, Konstantinos and Devetsikiotis, Michael. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.
- [11] Cohen, E. and Fiat, A. and Kaplan, H. Associative search in peer to peer networks: harnessing latent semantics. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, volume 2, pages 1261–1271 vol.2, 2003.

- [12] Communication Systems Group (CSG). Application of Blockchain Technology in the Swiss Cheese Supply Chain (CheeseChain), 2022. <https://www.csg.uzh.ch/csg/en/research/CheeseChain.html>, Last visit March 24, 2022.
- [13] ConsenSys Software Inc. Truffle suite, 2022. <https://trufflesuite.com/ganache/>, Last visit July 26, 2022.
- [14] Dan’s Tools. Unix timestamp. <https://www.unixtimestamp.com/>, Last visit July 16, 2022.
- [15] Davd Diener. Cheesechain, 2022. <https://github.com/Dave5252/BA-Code>, Last visit July 21, 2022.
- [16] Etheruem.org. Ethereum, 2022. <https://ethereum.org/en/learn/>, Last visit July 22, 2022.
- [17] Gargolinski Jaeger, Laura. Public versus private: What to know before getting started with blockchain, 2018. <https://ibm.co/2BZ8P7J>, Last visit March 25, 2022.
- [18] Girardi, Luca. Applikation der Blockchain-Technologie zur Authentifizierung von Tête de Moine AOP, December 2021. Bachelor Thesis, Hochschule für Agrar-, Forst- und Lebensmittelwissenschaften Bern.
- [19] Gu, Weiwei and Li, Jianan and Tang, Zekai. A survey on consensus mechanisms for blockchain technology. In *2021 International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA)*, pages 46–49, 2021.
- [20] Huang, Yuxin and Wang, Ben and Wang, Yinggui. Mresearch on ethereum private blockchain multi-nodes platform. In *2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 369–372, 2020.
- [21] Jayabalan, Jayapriya and N, Jeyanthi. A study on distributed consensus protocols and algorithms: The backbone of blockchain networks. In *2021 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–10, 2021.
- [22] Juma, Hussam and Shaalan, Khaled and Kamel, Ibrahim. A Survey on Using Blockchain in Trade Supply Chain Solutions. *IEEE Access*, 7:184115–184132, 2019.
- [23] Khan, Chris and Lewis, Antony and Rutland, Emily and Wan, Clemens and Rutter, Kevin and Thompson, Clark. A distributed-ledger consortium model for collaborative innovation. *Computer*, 50(9):29–37, 2017.
- [24] Krstić, Marija and Krstić, Lazar. Hyperledger Frameworks with a Special Focus on Hyperledger Fabric. *Vojnotehnicki glasnik*, 68:639–663, 07 2020.
- [25] Min, Hokey. Supply chain modeling: Past, present and future. *Computers & Industrial Engineering*, 43:231–249, 07 2002.
- [26] Mišić, Vojislav B. and Mišić, Jelena and Chang, Xiaolin. On forks and fork characteristics in a bitcoin-like distribution network. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 212–219, 2019.

- [27] Mohamed Ben-Daya and Elkafi Hassini and Zied Bahroun. Internet of things and supply chain management: a literature review. *International Journal of Production Research*, 57(15-16):4719–4742, 2019.
- [28] Monrat, Ahmed Afif and Schelen, Olov and Andersson, Karl. Performance evaluation of permissioned blockchain platforms. In *2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pages 1–8, 2020.
- [29] Nakamoto, Satoshi. What is the byzantine generals problem?, 2009. <http://www.bitcoin.org/bitcoin.pdf>, Last visit March 16, 2022.
- [30] National Institute of Standards and Technology. *Blockchain Technology Overview - NISTIR 8202: Draft January 2018*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.
- [31] Petri Helo and A.H.M. Shamsuzzoha. Real-time supply chain - A Blockchain Architecture for Project Deliveries. *Robotics and Computer-Integrated Manufacturing*, 63:101909, 2020.
- [32] Piper Merriam, Jason Carver. Web3.py - Introduction, 2022. <https://web3py.readthedocs.io/en/stable/>, Last visit July 26, 2022.
- [33] Python Software Foundation. time - Time access and Conversions, 2022. <https://docs.python.org/3/library/time.html>, Last visit July 28, 2022.
- [34] Remix team. Deploy & run transactions in the blockchain remix ide, 2022. <https://remix-project.org/>, Last visit July 26, 2022.
- [35] Ripeanu, M. Peer-to-peer architecture case study: Gnutella network. In *Proceedings First International Conference on Peer-to-Peer Computing*, pages 99–100, 2001.
- [36] Salah, Khaled and Nizamuddin, Nishara and Jayaraman, Raja and Omar, Mohammad. Blockchain-based soybean traceability in agricultural supply chain. *IEEE Access*, 7:73295–73305, 2019.
- [37] Samuel, Cyril Naves and Glock, Severine and Verdier, François and Guitton-Ouhamou, Patricia. Choice of ethereum clients for private blockchain: Assessment from proof of authority perspective. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5, 2021.
- [38] Scheid, Eder J. and Lakic, Daniel and Rodrigues, Bruno B. and Stiller, Burkhard. Plebeus: A policy-based blockchain selection framework. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8. IEEE Press, 2020.
- [39] Shuchih Ernest Chang and Yi-Chian Chen and Ming-Fang Lu. Supply chain re-engineering using blockchain technology: A case of smart contract based tracking process. *Technological Forecasting and Social Change*, 144:1–11, 2019.
- [40] Smith, Corwin. ORACLES, 2021. <https://ethereum.org/en/developers/docs/oracles/#:~:text=An%20oracle%20is%20a%20bridge,out%20to%20the%20real%20world.>, Last visit March 22, 2022.

- [41] Subramanian, Ganesan and Thampy, Anand Sreekantan. Implementation of hybrid blockchain in a pre-owned electric vehicle supply chain. *IEEE Access*, 9:82435–82454, 2021.
- [42] Suvitha, M and Subha, R. A survey on smart contract platforms and features. In *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 1, pages 1536–1539, 2021.
- [43] Swan, Melanie. *Blockchain : blueprint for a new economy*. O'Reilly Media, Sebastopol, Calif., 2015.
- [44] Umamaheswari, S. and Sreeram, Sruthi and Kritika, N. and Jyothi Prasanth, D. R. Biot: Blockchain based iot for agriculture. In *2019 11th International Conference on Advanced Computing (ICoAC)*, pages 324–327, 2019.
- [45] Vo, Khoa Tan and Nguyen-Thi, Anh-Thu and Nguyen-Hoang, Tu-Anh. Building sustainable food supply chain management system based on hyperledger fabric blockchain. In *2021 15th International Conference on Advanced Computing and Applications (ACOMP)*, pages 9–16, 2021.
- [46] Vujičić, Dejan and Jagodić, Dijana and Randić, Siniša. Blockchain technology, bitcoin, and ethereum: A brief overview. In *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, 2018.
- [47] Wikipedia. Byzantine fault, 2009. https://en.wikipedia.org/wiki/Byzantine_fault, Last visit March 21, 2022.
- [48] Yang, Jing and Wakefield, Ron and Lyu, Sainan and Jayasuriya, Sajani and Han, Fengling and Yi, Xun and Yang, Terry and Amarasinghe, Gayashan and Chen, Shiping. Public and private blockchain in construction business process and information integration. *Automation in Construction*, 118:21, 2020.
- [49] Zeng, Xueyun and Hao, Ninghua and Zheng, Junchen and Xu, Xuening. A consortium blockchain paradigm on hyperledger-based peer-to-peer lending system. *China Communications*, 16(8):38–50, 2019.
- [50] Zerocap, Victor Bastos. Decentralised peer to peer: Bitcoin's most important feature), 2021. <https://zerocap.com/decentralised-peer-to-peer-bitcoin/>, Last visit March 16, 2022.
- [51] Zheng, Zibin and Xie, Shaoan and Dai, Hongning and Chen, Xiangping and Wang, Huaimin. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, 2017.

Abbreviations

API	Application Programming Interface
ABI	Application Binary Interface
BC	Blockchain
BTF	Byzantine-Fault-Tolerance
CSG	Communication Systems Group
DAPP	Decentralized application
DB2BC	Database to Blockchain
DLT	Distributed Ledger Technology
DPoS	Delegated Proof of Stake
EVM	Ethereum Virtual Machine
IDE	Integrated Development Environment
POC	Proof of Concept
PoS	Proof of Stake
PoW	Proof of Work
SC	Smart Contract
SCT	Supply Chain Tracking

List of Figures

2.1	Linked Blocks forming a Blockchain[51]	6
2.2	Inner structure of a Block[51]	7
3.1	Traditional Supply chain process [25]	15
3.2	Scalability Behavior Testing [37]	17
4.1	Architecture of the overall CheeseChain solution	19
4.2	Data-workflow	20
4.3	Class diagram of the Smart Contract	22
4.4	Illustration of linkage between a file and its past versions on the BC	22
4.5	File Format Stored in the Server	23
5.1	Illustration of time usage for BC interaction, fetching, and parsing	36
5.2	Scalability of the solution	37

List of Tables

5.1	Performance Test Results	35
-----	------------------------------------	----

Appendix A

GraphQL queries

```
1 query DocumentAnswers($id: ID!) {
2   allDocuments(filter: [{id: $id}]) {
3     edges {
4       node {
5         id
6         form {
7           id
8           slug
9           __typename
10        }
11        workItem {
12          id
13          __typename
14        }
15        case {
16          id
17          workItems {
18            edges {
19              node {
20                id
21                task {
22                  id
23                  __typename
24                }
25              }
26            }
27            answers {
28              edges {
29                node {
30                  ...FieldAnswer
31                }
32              }
33            }
34          }
35        }
36        fragment SimpleQuestion on Question {
37          id
38          slug
39          label
40          isRequired
41          isHidden
42          meta
43          infoText
44          ... on TextQuestion {
45            textMinLength: minLength
46            textMaxLength: maxLength
47            textDefaultAnswer: defaultAnswer {
48              id
49              value
50              __typename
51            }
52          }
53          ...
54          placeholder
55        }
56      }
57    }
58  }
59 }
```

```

51   __typename
52 }
53 ... on ChoiceQuestion {
54   choiceOptions: options {
55     edges {
56       node {
57         id
58         slug
59         label
60         isArchived
61         __typename
62       }
63     }
64   }
65   __typename
66 }
67 choiceDefaultAnswer: defaultAnswer {
68   id
69   value
70   __typename
71 }
72 __typename
73 }
74 ...
75
76 fragment FieldTableQuestion on Question {
77   id
78   ... on TableQuestion {
79     rowForm {
80       id
81       slug
82       questions {
83         edges {
84           node {
85             ... SimpleQuestion
86           }
87         }
88       tableDefaultAnswer: defaultAnswer {
89         id
90         value {
91           id
92           answers {
93             edges {
94               node {
95                 id
96                 question {
97                   id
98                   slug
99                   __typename
100                 }
101               }
102             }
103             ... on StringAnswer {
104               stringValue: value
105               __typename
106             }
107             ... on IntegerAnswer {
108               integerValue: value
109               __typename
110             }
111             ... on FloatAnswer {
112               floatValue: value
113               __typename
114             }
115             ... on ListAnswer {
116               listValue: value
117               __typename
118             }
119             ... on DateAnswer {
120               dateValue: value
121               __typename

```



```

122
123 fragment FieldQuestion on Question {
124   id
125   ...SimpleQuestion
126   ...FieldTableQuestion
127   ... on FormQuestion {
128     ...
129   }
130
131 fragment SimpleAnswer on Answer {
132   id
133   question {
134     id
135     slug
136     __typename
137   }
138   ... on StringAnswer {
139     stringValue: value
140     __typename
141   }
142   ...
143   ... on FileAnswer {
144     fileValue: value {
145       id
146       uploadUrl
147       downloadUrl
148       metadata
149       name
150       __typename
151     }
152     __typename
153   }
154   ... on DateAnswer {
155     dateValue: value
156     __typename
157   }
158   __typename
159 }
160
161 fragment FieldAnswer on Answer {
162   id
163   ...SimpleAnswer
164   ... on TableAnswer {
165     tableValue: value {
166       id
167       form {
168         id
169         slug
170         questions {
171           edges {
172             node {
173               ...FieldQuestion
174             }
175           }
176         }
177         answers {
178           edges {
179             node {
180               ...SimpleAnswer
181             }
182           }
183         }
184       }
185     }
186     __typename
187   }
188   __typename
189 }

```

Listing A.1: Get answers of a form by its ID

Appendix B

Installation Guidelines

The following section describes how to setup the environment to be able to run the main.py in the CheeseChainPrivate folder. Note that for different operating systems the commands are slightly different.

B.1 Set up

1. Download Ganache from the truffle website: <https://trufflesuite.com/ganache/>
2. Clone the project from:

```
git clone https://github.com/Dave5252/CheeseChainPrivate.git
```

3. Install the necessary packages;

```
pip install -r requirements.txt
```

4. Copy paste the Smart Contract from the CheeseChainPrivate directory into the remix online IDE (<https://remix.ethereum.org/>)

B.2 Run

To run the main file the SC needs to be deployed and the refresh token needs to be extracted from the fromarte webpage.

1. Quickstart an Ethereum BC on Ganache
2. Copy the **RPC SERVER** address
3. Compile the SC on remix

4. Select Ganache as an Environment, paste in the **RPC SERVER** address.
5. Deploy the SC
6. On Ganache in transactions select the deployed SC and copy the **SENDER ADDRESS** and **CREATED CONTRACT ADDRESS** into the corresponding variables in the config.json (`["config"]["blockchain"]`) file in the directory.
7. (Only if the refresh token is invalid. *I.e.*, after logging in with the account on a different host) Login into Fromarte, with the Developer Tools Interface (F12) open on the network tab and copy the refresh token from the **token** into the Config.json file (`["config"]["fromarte"]`).
8. Run the main.py file

B.3 Synchronization

The files are synchronized onto the host and the BC, with the ID (e.g., RG9jdW1lbnQ6MDg2Yzc3ODAtNDZlYS00Y2IyLTlhMWQtMjliZmUOMzQ2NWYy) forms can be fetched from the BC through the remix IDE. In the folder *BackupFiles* the corresponding files with the relevant information can be found, as well as all **RUNNING** forms in the *BackUp* file in the main directory.

Appendix C

Contents of the ZIP

- PDF Of the Thesis
- LaTeX source code
- Midterm presentation in PPTX
- Source code of the solution