Case study **34**

Blockchains

In this study, we examine some fundamental aspects of blockchains, particularly the security of data and the way(s) it is achieved through cryptographic transformations. Basically, a blockchain is a historical database in which the description of operations, generally called transactions, are stored in chronological order. Once recorded, the data of a transaction can never be deleted nor modified.

The document first introduces the elements of cryptography necessary to build a blockchain, notably secure hashing, and symmetric and asymmetric key encryption. Then, it describes the distinctive aspects of blockchains independently of its application domain and applies them to cryptocurrencies. Finally an experimental toolbox, comprising a collection of functions designed to manage and explore blockchains, is built step by step.

Keywords. blockchain, blockchain explorer, proof of work, distributed database, cryptocurrency, trust, security, cryptography, RSA, AES, secure hashing

34.1 Introduction

Blockchain is certainly one of the most popular techniques currently discussed, both in the general public and in scientific circles. Initially known as a cryptocurrency implementation platform, its applications have quickly spread beyond the financial world. Now applied (or suggested to apply) to the most diverse fields of human

activity, blockchain is considered by some futurologists as **the** disruptive technology that can lead to major societal changes.

Far from these speculations, whose validity has yet to be demonstrated, this study has a particularly modest objective: to illustrate in practice the technical bases of blockchain technology.

In short, a blockchain will be seen a a *massively replicated, secure, append-only, historical database* supported by a peer-to-peer network. The complexity of its data structures and management rules makes blockchain a remarkable application of database technology. That alone justifies a full chapter in this series of case studies

Since blockchains make intensive use of cryptographic operations, we give in Section 34.2 a very short introduction of those used in this technology. In Section 34.3 we recall the principles of blockchains, that we apply to cryptocurrencies in Section 34.4. The remaining of this study (from Section 34.5) is devoted to the analysis and implementation of some of the main functions of a didactic blockchain management system.

34.2 Elements of cryptography

Cryptography is a set of techniques devoted to the transformation of explicit information into a meaningless string of bits. Their main objective is to **securely** store or communicate information. The concept of *security* encompasses several issues, such as preventing unauthorized third parties to read the information, checking its integrity or preventing parties to contest its authenticity. We will briefly describe the principles of three important cryptographic techniques used by blockchains. In this presentation, we will call *message* any kind of source user data that we intend to process through cryptographic techniques: text, file, picture, program, etc.

Secure hashing

Hashing is the derivation of a (seemingly) random fixed-length bit string from a source message. This derivation in performed by a hash function (let us call it **h**) that implements complex mathematical operations. The resulting bit string is called the **hash** or the **digest** of the message. A good hash function must satisfy the following properties:

1. **h** is deterministic: applying **h** to source message **M** always provides the same result.
2. **h** is not invertible: there is no way to compute message **M** from its hash. In other words there is no function **g** such that **M** = **g**(**h**(**M**)).
3. **h** is injective: applying **h** to two different messages yields two distinct hashes: if **M1** ≠ **M2**, then **h**(**M1**) ≠ **h**(**M2**). This property is also called *collision resistance*.

One of the most popular hash function is SHA256, that produces hashes of 256 bits. It is available in the SQLfast library under the name hash(). The fact that (good) hash functions have no inverse is of course important but it does not prevent attackers to try to *crack* a hash, that is to guess the (unknown) source message of a (known) hash through a brute-force attack such as that of Script 34.1, where messages(N) is an iterator that yields successive bit strings of length **N**, from 'ob0000...0000' to 'ob1111...1111', and Hash is the hash of which one tries to guess the source.

```
def guessSource(Hash):
    # Try messages of length 1 to 1000
    for N in range(1,1001):
        # Create an iterable object from class 'messages'
        Nmessages = messages(N)
        for source in Nmessages:
            if hash(source) == Hash:
                return source
    return ''
```

Script 34.1 - Guessing the source message of a known hash through a brute-force method

Three comments on this approach:

– The number of potential messages to check is extremely large: 2^N for each value of **N** (generally, we do not know the length of the actual message), so that the exploration is not feasable with current computing technology. A machine capable to compute *one billion hashes per second* would take about 300 years to examine all the combinations of 64 bits, i.e., about 10 printable ASCII characters.

– Property 3 defines *perfect* hash functions. Actually, there is no way to ensure that, therefore, there is no guarantee that the source message found with this method is the true one, intended by the sender. Good functions make collisions (the fact that two distinct messages are assigned the same hash) extremely rare, in such a way that they are considered *practically* injective. Such good functions are described as *secure*, hence the name *secure hash algorithm* or SHA.

– More refined cracking techniques could be based on the knowledge of parts of the original message, such as its length or a constant signature or introduction. However, a property of secure hash functions makes such hints useless: two almost identical source messages produce very different hashes.

  This is illustrated by the following experiment: computing the hash of these two messages, where only the case of the first letter is different,

```
compute hash1 = hash('what is the best database language?');
compute hash2 = hash('What is the best database language?');
write $hash1$;
write $hash2$;
```

produces these very different hashes (displayed in hexadecimal):

```
e6b67be466c63231af78a4bece714def58f33cbc48911b7319579cee242b0415
d9879a0017d31dd03144ce0f4be8e0f1b46d503c27c1ee4c87c49d4ee49548c1
```

It is important to understand that hashing does not preserve the contents of the source message, contrary to the next two techniques.

Symmetric key cryptography

Encryption is a reversible transformation of a source message that produces a meaningless byte (or character) string. This transformation is performed by a complex algorithm based on an *encryption key*. The encrypted message can be securely transmitted or stored. To recover the source message, the encrypted message must in turn be transformed by a decryption algorithm using an *decryption key*.

In *symmetric key cryptography* the encryption and decryption keys are the same. To protect the encrypted messages, this key must be kept secret, only known by the sender and the recipient(s) of the messages.

The SQLfastUDFlib library includes two primitives to encrypt and decrypt messages. They are particularly fit to process plain character strings. The following script illustrates their usage:

```
set message0  = What is the best database language?;
set secretkey = my secret key;
compute crypto   = encrypt('§message0§','§secretkey§');
compute message1 = decrypt('§crypto§','§secretkey§');
write $message0$;
write $crypto$;
write $message1$;
```

With the following result, showing that the original message has been preserved:

```
What is the best database language?
xOGB54XM5YXoiNCF29LslJPJxObG1oHeypnZ2o7a2sTZyrM=
What is the best database language?
```

Formally, if **M** is the source message and **K** the symmetric key:

```
decrypt(encrypt(M,K),K) = M
```

To process pure binary messages, functions generateSYMkey, encryptSYM and decryptSYM of LStr.py library are more appropriate. Function generateSYMkey generate a pseudo-random byte secret key of given length (expressed in base64). The usage of these functions is illustrated here below. The length of the key is 12 bytes (i.e., exactly 16 in base64).

```
set message0  = What is the best database language?;
 function secretkey = LStr:generateSYMkey 12;
 function crypto    = LStr:encryptSYM  {$message0$},$secretkey$;
 function message1 = LStr:decryptSYM  {$crypto$},$secretkey$;
 write $message0$;
 write $secretkey$;
 write $crypto$;
 write $message1$;
```

With the expected result:

```
What is the best database language?
ue9ihdMB7hnUC/sn
zM2a3YjNwGKr0NN1pZTm4pXJmt3Jxq61nIjatrGW6M_cyng=
What is the best database language?
```

The SQLfast functions are based on the *Vigenère* symmetric key algorithm. This technique is particularly simple and the encryped messages are reasonably hard to crack. More secure algorithms are available such as the *Advanced Encryption Standard* (AES). The main weakness of symmetric cryptography is that all the senders and recipients must keep the key secret. Once the secret key has been generated, it must be distributed over a secure channel, for example through asymmetric encryption.

Asymmetric public key cryptography

Asymmetric encryption uses two keys. The *public key* is used by all the senders who intend to send messages to a recipient. This key is specific to the recipient. It is public and can be found in some sort of public directory. When the recipient receives a message that has been encrypted by her public key, she can decrypt it with a second key, her *private key*. Only the recipient possesses this private key, which is then easier to protect.

To perform asymmetric encryption, SQLfast offers three functions similar to those of symmetric cryptography: generateRSAkey (that yields both public and private keys), encryptRSA and decryptRSA of LStr.py library. They use RSA, among the most popular asymmetric cryptographic algorithms. The usage of these functions is illustrated in the following snippet. The length of the public key is 420 bits (converted in base64).

```
set message0  = What is the best database language?;
function secretkey,publickey = LStr:generateRSAkeys 420;
function crypto    = LStr:encryptRSA {$message0$},$secretkey$;
function message1 = LStr:decryptRSA {$crypto$},$publickey$;
```

Formally, if **M** is the source message, **P** the public key and **V** the private key:

```
decryptRSA(encryptRSA(M,P),V) = M
```

A major drawback of the asymmetric technique is that the public key must be at least as long as the longest message to encrypt. To overcome this problem, one can encrypt long messages by slices the length of the key or to use symmetric encryption and to encrypt the symmetric key through an asymmetric technique.

Blockchain management makes intensive use of asymmetric cryptography to verify the authenticity of transactions. However, the roles of *sender* and *recipient* are inverted. Only the sender is allowed to encrypt a message but all the members of the blockchain, the recipients, may decrypt it. So, when a pair of (*public*, *private*) keys are generated by the RSA algorithm, the originally *public key* is used as a *secret key*, which must be carefully protected by the sender, while the originally *private key* is made *public*, allowing everybody to decrypt messages. To summarize:

standard	blockchain
public key	secret key
private key	public key

34.3 The blockckain paradigm

Basically, considered as a service, a blockchain is a *journal*, or a *ledger*, in which a community of members securely record information on operations of interest to them. Quite often, these operations concern the creation, modification and exchange of critical resources such as money, energy, goods, services, stock market securities or real estates.

In the (so far *usual*) real world, such operations are controlled by trusted intermediaries such as banks, public administrations, stock exchange authorities, lawyers or notaries public.

The most significant characteristic of blockchains is that there is no such intermediaries in charge of controlling the legitimacy of the operations. Each member is responsible for the validity of her operations, but also of the validity of the operations carried out by all the other members of the community.

This rises the issue of **trust**. As a member of the community, how can I be certain of the **validity of an operation**? When a house is being sold, is its seller really the owner this house? When a customer pays me the product I sent her, is the balance of her account sufficient?

In addition to the validity of the operations themselves, I must be confident that the information recorded in the ledger will never be falsified; in other words, that the content of the blockchain never can be altered once it has been recorded (it must be **immutable**).

The trust in the information of a blockchain is ensured by a set of rules that define the way operations must be recorded. These rules form the protocol of this blockchain. Simply stated, the blockchain technology must meet the following conditions:

– *Reliability*: the operations described in the blockchain are definitively effective as soon as they have undergone a validation process.
– *Non falsifiability*: data can be appended to the blockchain, but, once recorded, they can no longer be deleted or modified.
– *Transparency*: the recorded data are public; all members of the community can review them and check their validity.
– *Robustness*: the blockchain must resist all kinds of attack, be they accidental or intentional.
– *Anonymity*: the identity of the members involved in an operation is protected, except in some special situations, such as transactions involving a bank (to prevent from money laundering), this identity is mentioned via a meaningless pseudonym.

The absence of a trusted third party has an important consequence: there is no central service responsible for storing the blockchain. It is simply copied on the computer of each member of the community, thus contributing to its robustness.

Members

A member who joins the community receives three specific information items:

– an unique *identifier*, which is a random character string (principle of *anonymity*) that will be used to identify her in any transaction she will be a partner of,
– a *secret key* with which she will *encrypt* her data,
– a *public key* with which all the members of the community can *decrypt* these data.

These keys allow data to be securely recorded in the blockchain through asymmetric cryptography. The couples (*identifier*, *public key*) are public and can be known by the community, or at least by a part of it.

Transactions

Each operation related to the resource managed by the blockchain is executed as a *transaction*. The properties of the transaction are recorded in the blockchain as soon as it has been executed, but it will be effective only when it is validated.

Let us suppose that I sell my house to a buyer for 150,000€, the data of this transaction are recorded in the blockchain: my id, the id of the buyer, the id of the house, the price we agree upon and the date. The sale will be effective when these data have been validated: I am the owner of the house, the house is free from any debt, the price is plausible, the buyer exists and this amount is available on her account.

When the data of the transaction have been recorded in the blockchain, they are vulnerable, notably prone to all kinds of fraud, such as a member discretely modi-

fying some data. For example, my buyer could be tempted to lower the value of the house to, say, 125,000€. To prevent this fraud, I modify the transaction in this way:

– When all the data of the transaction have been collected (let us call them **T**), I encrypt them with my secret key (**S**); the result is the *signature* of the transaction:

  signature = encrypt(**T**,**S**)

– I add this signature to the transaction data.

– I make this augmented transaction public.

Therefore, every member of the community can examine this transaction, decrypt its signature, compare the result with the source data and conclude that the source data are authentic or corrupt. Denoting by **P** the public key associated to **S**, **T** is considered correct if:

  decrypt(**signature**,**P**) = **T**

Since only the creator of the transaction can encrypt the data, there is no way for a fraudster to alter the source data.

To summarize,

– a transaction describes an operation,

– it is complemented by its signature

– it is distributed to the members of the blockchain,

– who can validate it,

then it becomes effective (or rejected). However, as we will see, recording a transaction in the blockchain is a bit more sophisticated.

Blocks and blockchains

For reasons that will be discussed later, when a sufficient number of transactions have been created, they are grouped together to form a **block**. This block is assigned an identifier, that becomes a part of it, as well as the identifier of the last block that has been created. So, each block (but the first one) references the previous block, thus forming a chain of blocks, hence the term **blockchain**.

The identifier of a block is computed as the *hash* of its contents, that is:

$$\texttt{Bbody} = \texttt{trans}_1 + \texttt{trans}_2 + \ldots + \texttt{trans}_n + \texttt{rectime} + \texttt{previous}$$
$$\texttt{Bhash} = \texttt{hash(Bbody)}$$
$$\texttt{block} = \texttt{Bhash} + \texttt{Bbody}$$

where $\texttt{trans}_i$ is the i^{th} transaction collected since the creation of the preceeding block,
n is the standard capacity of a block,

`rectime` is the timestamp of the block,
`previous` is the hash of the previous block,
`Bhash` is the identifier of the block being built,

The quality of the hash function, based on SHA256, ensures that there will be no collision between two blocks, at least in a forseeable future!

The block structure provides a second level of security against data corruption, whether accidental or intentional.

Suppose that the creator of a transaction wants to fraudulently modify it, for example to increase the price of the house he sold. Since he possesses the secrete key of the transaction, he can modify the transaction data, recalculate the signature then replace the transaction in its block, hoping that no one will notice.

Unfortunately, the block structure makes the task a bit more complex. The hash of the block now is invalid and must be recalculated then replaced in its block. Therefore, the next block becomes invalid, since it references the fraudulent block. It must be updated as well, and so on until the last block of the chain has been modified. Worse, all the members of the community have a copy of these blocks. The fraudster therefore has no choice but to ask them to replace these blocks with the new ones. Hoping that nobody will notice!

Validation of a block

The last step of the protocol consists in validating the block we have built and publishing it to make it accessible to all the members of the community. This task is fairly tricky and will be best explained in the specific application domain of cryptocurrencies.

34.4 Application: cryptocurrencies

One of the most popular applications of blockchains is *cryptocurrencies*, and more specifically *bitcoins*, managed through the *Bitcoin* protocol. When I want to buy a book that cost 20 €, I usually transfer this amount of money from my account to that of the bookshop. For this, I make use of the services provided by a bank, which both (me and the bookshop) consider a *trusted intermediary.*

The objective of the Bitcoin protocol is to allow us to execute such operations without relying on the services of a bank or of any other intermediary such as PayPal. If the bookseller and myself are members of the same cryptocurrency community, we just have to record a *money transfer transaction* in the blockchain of our community.

Following the description given in the preceeding section, such a transaction will normally include *my id*, the *id of the bookseller*, the *price* of the book and the *date* of the transaction. And of course the *signature* of these data.

Note. In the following, we describe a protocol similar to that of Bitcoin but strongly simplified, in order to comply with the didactic objective of this series of case studies. Motivated readers are invited to refine this protocol to include some specific aspects of Bitcoin, such as *multi-output transactions* and *miner conflicts.*

The member id

When joining the blockchain system, each member receives a *public key* through which all the members can decrypt the signature of his transactions. This key is unique and therefore could be used to identify the member. However, the public key is a fairly large character string, so that a new, shorter, identifier is derived from it by hashing this key. In the Bitcoin protocol, the member id, called *Bitcoin address*, is 160 bit long.

The transactions

First of all, some terminological clarification: in addition to *member* and *member id*, we will use the terms *account* and *account id*. A member is the agent who creates and manages an account. There is a one-to-one relationship between members and accounts: each member manages one account and each account is managed by one member. However, the blockchain only knows accounts. We will keep the concept of member when necessary, notably to give operations an intuitive interpretation.

To make things concrete, we consider three types of transactions.

- **registering** a new account. *Input*: recording date. *Output*: an account id, a secret key and a public key.
- **depositing** an amount of money on an account. *Input*: recording date, id of the source account and the amount deposited.
- **transferring** an amount of money from a source account to another account. *Input*: recording date, id of the source account, id of the recipient account and the amount transferred.

When recording a transaction, we add to its base data a *transaction id* and its *signature*, computed with the secret key of the member who initiated the operation.

Block building

When a transaction has been recorded, it is sent to the members of the blockchain who are in charge of validating them. These members are called **miners**, for a reason we will explain below.

First, miners verify that the data of each transaction are not corrupt by comparing them with the data of the decrypted signature. Then, they analyze the components of the transactions to verify that they are semantically correct: for example, the accounts ids reference accounts that really exist and the amount of money is available, therefore preventing a member to spend more than the balance of his account.

When the transactions a miner has received are proved to be valid, they are grouped to form a block. This block is signed with its hash and sent to the other members of the blockchain to (tentatively) become its new last block. This step is quite fast and does not require powerful machines.

Finally, we note that the hash of a block is guaranteed to be unique and therefore can be used as the **identifier** of the block in its blockchain.

Block mining

A first problem that arises is that all the miners carry out the same task, so that many candidate blocks are broadcasted to the community almost at the same time. Hence the dilemma: which one to choose? In a context in which there is no central authority, the idea is simple, the first one is the winner. To make this rule effective, the process of block building must be slowed down, in such a way that the candidate blocks are received in a (seemingly) random way.

Another problem is that, due to the ease with which this task can be carried out, quite many members can pretend to the status of miner, therefore opening the door to the risk of inappropriate behaviour and fraud. It is important to limit the number of miners to those who really are interested in, and therefore willing to contribute to, the proper functioning of the blockchain. Therefore, the miners are required to prove that they enjoy a certain level of resource that gives them an outstanding status among the community that, in turn, grant them its trust. The most popular of these resources are their *computing power* (the miner must show his *Proof of work*) and their *notoriety*[1] (the miner must show his *Proof of stake*). We will describe the first of them.

To solve these problems, the blockchain protocol imposes a special constraint on the hash with which the block is signed: the block must be accompanied by a short bit string, called the **nonce** of the block, that, if appended to the block, produced a hash the **k** first bits of which are zeroes.[2] Satisfying this constraint consists in finding a nonce. There is no clever way to find it: one just has to try all the combinations of **k** bits until one of them produces a good hash. That is:

```
body = transactions + rectime + previous
nonce is valid if
   hash(body + nonce)[0:k] = '0b000...000'
```

This nonce and the hash it yields are added to the block, which is then sent to all the members as the suggested last block of the chain. Figure 34.1 illustrates the structure of a blockchain by showing the contents of two successive blocks. In each block, Bhash denotes the valid hash, used as the block identifier.

Checking that blocks are valid is quite fast, so that every member, whether miner ou not, can verify that a block is not corrupt. On the contrary, finding a valid nonce

1. The notion of *notoriety* can be quite diverse: the current balance of your account, the average balance in the last year, your activity in the blockchain, your social position, etc.
2. In 2019, k = 72

is very expensive in computing time. In particular, it requires a large computing infrastructure. So, broadcasting a block with a valid nonce is a proof that one has worked *very hard*. In other words, a valid nonce is a **proof of work** that any other member can verify.

To make things even more challenging, only the first miner who finds a valid nonce wins the competition. We could wander what may motivate miners to perform such a hard and expensive work for such an uncertain result. The answer is both simple and universal: *money*! The winner receives a certain amount of crypto-money as compensation for his work. Hence the name of *miner*: working hard to hopefully find, from time to time, some small golden nugget!

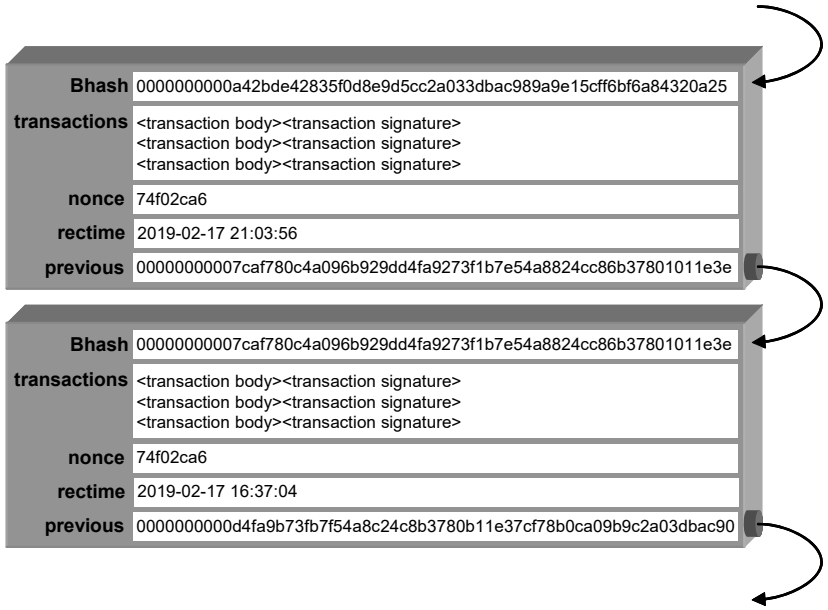

Figure 34.1 - Fragment of a blockchain

This description of the blockchain mechanisms is just an introduction to the main principles. It leaves out many problems and their solution. More details can be found in countless publications both in libraries and on the web. As far as this study is concerned, we now know enough to start the implementation of some components of a simplified prototype blockchain.

34.5 The BLOCKCHAIN toolbox

The programs that implement a blockchain system *in the real world* form a complex distributed system. In the context of this series of case studies, the objective of which basically is didactic, it would not be realistic to develop a full-fledged block-

chain software. We will rather develop a set of independent functions sharing a database representing an elementary but representative blockchain.

The application chosen is that of a cryptocurrency based on the presentation of Section 34.4. The function set includes the three basic transactions of **registering an account**, **depositing an amount** of money and **transferring an amount** of money from a source account to a recipient account. These transactions will then be processed by a **validation** function and gathered into **blocks**, including their **mining**. To these functions, that form the basis of a blockchain application, we add a set of **visualization** functions that will allow us to examine the data with different degrees of granularity.

The basic mining function will be discussed and implemented, but we will leave aside the aspects of *distribution* (data replication in each member machine in particular), *reward* (remuneration of miners), *competition* (the first miner "*takes it all*") or *conflict resolution*. Similarly, transfer transactions between accounts are limited to a single recipient account (no *multiple ouputs*). These extensions can be developped without problem once a solid basic architecture has been defined.

Following the approach of the case studies series, a blockchain will be designed and implemented as a relational database application, naturally developed in the SQLfast language and environment.

34.6 The database

The main data structures include three public tables, containing information on the *transactions*, the *blocks* and the *public keys* of the members.

34.6.1 The transactions (table BTRANSACTION)

There are three types of transactions: registering a new account (Operation = REGISTER), depositing an amount of money on an account (Operation = DEPOSIT) and transferring an amount of money from a source account to a recipient account (Operation = TRANSFER) .

Let us consider the example of a transfer transaction of an amount of 100 units of money from **Mary**'s account to **Luke**'s account. Figure 34.2 shows how we will code this information. It indicates the *type of operation*, (Mary's) *source account id*, (Luke's) *recipient account id*, the *amount* of money transferred, the *recording date* of the transaction and finally the *signature* of the transaction (only a part of the signature is shown).

To this basic data, we add the *reference of the block* (its Bhash) in which the transaction will be inserted. Then we assign to each transaction an *id* (unique in the blockchain) formed by the id of the source account to which we add the recording date. This pattern is valid provided that the transactions initiated by an account are created on distinct dates, which is not a particularly hard constraint since this date is detailed to the second.

Operation: TRANSFER
Source: FT7UzFgSpMmNsnlet_ooAKV4wWTyJAaLrnYH6TRtIBw=
Recip: QWA7QEjrvrsg1lpALJi34f8Ksrzb0ANfa5tqsylm7Xw=
Amount:100
RecTime: 2019-05-23 15:58:3
Signature: C75d4HLpm9STKHj[..]h-blt2uCmMM_mfK

Figure 34.2 - Data describing a TRANSFER transaction

Finally, in order to *facilitate experimentation*, transactions will be provided with a short numerical identifier, called a *transaction key*, which will be assigned automatically. This identifier is obviously not part of the blockchain model.

These components are translated into the BTRANSACTION table[3], the structure of which is shown in Script 34.2. Column TransKey will contain the transaction key, TransID the transaction id, Bhash the reference of the block. The other columns have an obvious meaning.

```
create table BTRANSACTION(
   TransKey   integer      not null primary key autoincrement,
   TransID    varchar(72)  not null unique,
   Bhash      varchar(48)  references BLOCK(Bhash),
   Operation  char(12)     not null,
   Source     varchar(48),
   Recip      varchar(48),
   Amount     decimal(16,10),
   RecTime    datetime     not null,
   Signature  varchar(512) not null);
```

Script 34.2 - Schema of the table describing the transactions

34.6.2The blocks (table BLOCK)

A *miner* creates a block as soon as a sufficient number of candidate transactions have been received. Basically, a block consists of a *block identifier*, a sequence of *transactions*, its *creation date* and the identifier of the *previous block*.

The identifier of the block is obtained by hashing its content to which a **nonce** is added. This nonce will be represented in the BLOCK table as a long integer represented by a decimal character string. The hash is translated into base64 for convenience. Finding this nonce, which is particularly time-consuming, constitutes the *proof of work* of the block validation. In the standard version of the toolbox, k = 20, which requires a mining time from 5 to 30 seconds for blocks of 5 transactions.

3. 'TRANSACTION' would have been a better name. Unfortunately, it is an SQL reserved word!

The maximum number of hash computations is 5,000,000. These parameters can be changed if needed.

Transactions are not integrated into the block as usually described in the literature (see Figure 34.1) but are associated with it via foreign key Bhash in table BTRANSACTION. We will simply indicate the number of transactions associated with the block.

A block that has just been created is submitted to the community of miners. At this stage, it is only a project that has yet to be validated and selected as the official block. Blocks are characterized by their *status*, which indicates their validation status: 0 for a validated block and 1 for a selected block.

Just as we have done for transactions, once again to facilitate experimentation, we will associate a numerical identifier, called a *block key*, to the blocks. The translation into a table structure is illustrated in Figure 34.3. Column BlockKey is the block key, Bhash the hash of the block, and therefore its id, Transact is the number of transactions.

```
create table BLOCK(
   BlockKey   integer      not null primary key autoincrement,
   Bhash      varchar(48)  not null unique,
   Transact   integer      not null,
   Nonce      varchar(32),
   RecTime    datetime     not null,
   Status     integer      not null default 0,
   Bprev      varchar(48) references BLOCK(Bhash));
```

Script 34.3 - Schema of the table the blocs

34.6.3The public key directory (table DIRECTORY)

To transfer an amount of money to a recipient account, we are supposed to know its id, just as, to make a bank transfer, we need to know the number of the recipient account. To validate a transaction we must decrypt its signature to verify that it corresponds to the values of the different fields of this transaction. This decryption is performed by the public key of the creator of the transaction, which is the member who owns the source account.

We must therefore create a *directory* that, for each account id, provides its public key. This directory is stored in the DIRECTORY table whose schema is shown in Script 34.4. A column has been added indicating whether (1) or not (0) the account owner is a miner.

```
create table DIRECTORY(
   AccountID  varchar(64)   not null primary key,
   PublicKey  varchar(1400) not null,
   Miner      integer       not null default 0);
```

Script 34.4 - Schema of the table providing public keys

34.7 Experimental data

The data in the BTRANSACTION, BLOCK and DIRECTORY tables are public, and therefore accessible to all members of the blockchain.

To these so-called *functional data* (i.e., necessary for the operation of the blockchain), we must add those that will allow us to experiment with the concepts of blockchain. These data, which do not exist in real implementations (we represent them in red in the table schemas), allow us to experimentally perform, in a simple and intuitive way, the different operations on our block chain. We will call them *experimental data*.

First, it should be noted that tables BTRANSACTION and BLOCK already contain experimental data in the form of the transaction keys (TransKey column) and the block keys (BlockKey column).

The data of the accounts we will play with will be stored in ACCOUNT table. An account has an id generated by hashing its public key and displayed in base64 character code. Memorizing and manipulating these long and meaningless character strings will not be particularly comfortable. For this reason, we associate with each account a nickname easy to remember, for example a person name, which we call *private name* (PrivName column). So, we will describe a transfer operation in this way:

Mary *transfers an amount of 100 to* Luke*'s account*

obviously more natural than its technical equivalent:

an amount of 100 is transferred
from account `FT7UzFgSpMmNsnlet_ooAKV4wWTyJAaLrnYH6TRtIBw=`
to account `QWA7QEjrvrsg1lpALJi34f8Ksrzb0ANfa5tqsylm7Xw=`

When an account is created, a pair of keys are generated to compute (by the secret key) and then decrypt (by the public key) the signature of transactions. In a real implementation, only the owner of the account knows its secret key, which he must store and protect with the greatest care. In this experimental prototype, we will play all the roles: account creator, source, recipient and miner. We will therefore need to know the secret keys of all the members of the blockchain!

These experimental data are stored in table ACCOUNT. It contains the *account id*, the *private name*, the *secret key* and the *time* the data of the account were recorded

(Script 34.5). There is no need to store the public keys since they already are available in table DIRECTORY.

```
create table ACCOUNT(
   AccountID  varchar(64)   not null primary key,
   PrivName   varchar(64)   not null,
   SecretKey  varchar(360)  not null unique,
   RecTime    datetime      not null);
```

Script 34.5 - Schema of the table describing the accounts

34.8 Account registration

Creating an account leads, on the one hand, to the creation of an RSA key pair (secret key and public key) allowing its owner to execute transactions on this account and, on the other hand, to the creation of a transaction that formalizes the existence of this new account.

The whole operation comprises five steps, shown in Script 34.6:

```
ask name = Private name:;
function skey,pkey = LStr:generateRSAkeys 1700;
function aid = LStr:hash {$pkey$},2;
set      dat = $date$ $clock$;
set      tid  = $add$-$date$_$clock$;
set      op  = REGISTER;
function sig = LStr:encryptRSA {$tid$;$op$;$aid$;$dat$},{$skey$};
insert into ACCOUNT (AccountID,PrivName,SecretKey,RecTime)
       values ('$aid$','§name§','$skey$','$dat$');
insert into DIRECTORY (AccountID,PublicKey)
       values ('$aid$','$pkey$');
insert into BTRANSACTION
      (TransID,Operation,Source,RecTime,Signature)
       values ('$tid$','$op$','$aid$','$dat$','$sig$');
```

Script 34.6 - Script creating a new account (excerpts)

1. Computing the *secret* and *public keys*. We use function generateRSAkeys of LStr library. The size of the secret key must be greater than or equal to the size of the strings it must encrypt. A 1700-bit key is sufficient to encrypt a transfer transaction, the largest of the three types. The length of the secret key thus generated, expressed in base64, is 280 characters and that of the public key is about 1330 characters. These keys are stored in variables skey and pkey.
2. Computing the *account id*. It is obtained by hashing the public key using the hash function of LStr library. The second parameter of the function specifies

the format of the result: hexadecimal (1) or base64 (2). We choose the shorter base64 format. This value is computed in variable aid.

3. Inserting a row describing the account in table ACCOUNT. We record the *account id*, the *private name* the experimenter will associate with it, the *secret key* and the *date* the account was created. This date consists of the date itself followed by the current time of the day, information obtained by the date and clock functions.
4. Inserting a row publishing the *public key* associated with the *account id* in table DIRECTORY. By default, a new member is not a miner.
5. Inserting a row describing the registration transaction in table BTRANSACTION. The *transaction id* (column TransID, the value of which comes from variable tid) is made up of the account id concatenated with the creation date. The *signature* of the transaction is obtained by encrypting with the secret key the string formed by the transaction id, the transaction, the account address and the date the account was created. An example of contents of the new BTRANSACTION row is shown in Figure 34.3.

TransKey	259
TransID	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=-2019-06-11_09:32:17
Operation	REGISTER
Source	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=
Recip	--
Amount	--
RecTime	2019-06-11 09:32:17
Signature	Bp8tBqgkTgxdyig[..]PmWAK8q9ngBMKWf

Figure 34.3 - Composition of a row of table BTRANSACTION describing the creation of an account for member *Mary*

The operation of account creation is coded in script _BC_Create_Account.sql.

34.9 Amount deposit

To simplify, we postulate that the amount a member deposits on his account comes from a source that we agree to ignore. So, this account is the recipient of an amount from an unknown source.

The function requires the identification of the recipient account and the amount to deposit. In this prototype, the recipient account is identified by the private name of the member.

The logic of the function is similar to that of creating an account, except that the value of the secret key is not calculated but is extracted from table ACCOUNT.

The main part of the deposit function is shown in Script 34.7. Figure 34.4 represents an example of the contents of a deposit transaction.

```
ask aid,amnt = Your name:[!select PrivName,AccountID
                           from   ACCOUNT order by PrivName]
              |Amount:;
extract  skey = select SecretKey from ACCOUNT
                 where  AccountID = '$aid$';
set      dat = $date$ $clock$;
set      tid = $aid$-$date$_$clock$;
set      op  = DEPOSIT;
function sig = LStr:encryptRSA
                 {$tid$;$op$;$aid$;$amnt$;$dat$},{$skey$};
insert into BTRANSACTION
       (TransID,Operation,Recip,Amount,RecTime,Signature)
values ('$tid$','$op$','$aid$',$amnt$,'$dat$','$sig$');
```

Script 34.7 - Script executing a money deposit (excerpts)

TransKey	274
TransID	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=-2019-07-09_16:50:32
Operation	DEPOSIT
Source	--
Recip	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=
Amount	150
RecTime	2019-07-09 16:50:32
Signature	7h7Y025kG8tBqd[..]Pq9ngBMKWfmWAK8

Figure 34.4 - Composition of a row of table BTRANSACTION describing a DEPOSIT operation: *an amount of 150 is deposited on Mary's account*

The operation of amount deposit is coded in script _BC_Execute_Deposit.sql.

34.10 Money transfer

The user specifies the *source* account, the *recipient* account and the *amount* to be transferred between them. Accounts are designated by the user via their private names, which the ask function converts into account ids, then stored respectively in the variables **sou** (the *source* account) and **rec** (the *recipient* account) (Script 34.8).

The most interesting part of the function is the validation of the amount to be transferred, which cannot exceed the balance of the source account. This balance is obtained by a general procedure combining three subtotals representing the inputs and outputs of the source account:

– *Input*: the sum of the deposits on this account. This value is stored in variable **plusD**.

– *Input*: the sum of the transfers for which this account is the recipient is added. This value is stored in variable **plusT**.

– *Output*: the sum of the transfers for which this account is the source. This value is stored in variable **minus**.

We note the use of SQL function coalesce, that replaces *null* values with numeric *zero*. A *null* value appears when no rows have been found, which, in this particular case, must be interpreted as 0.

```
ask sou,rec,amnt =
          Your name:[!select PrivName,AccountID from ACCOUNT
                     order by PrivName]
         |Recipient:[!select PrivName,AccountID from ACCOUNT
                     order by PrivName]
         |Amount:;
extract skey = select SecretKey from ACCOUNT
               where  AccountID = '$sou$';
set     tid  = $sou$-$date$_$clock$;
set     op   = TRANSFER;
set     dat  = $date$ $clock$;
function sig = LStr:encryptRSA
              {$id$;$op$;$sou$;$rec$;$amnt$;$dat$},{$skey$};
extract plusD = select coalesce(sum(Amount),0)
                from   BTRANSACTION
                where  operation = 'DEPOSIT'
                and    Recip = '$sou$';
extract plusT = select coalesce(sum(Amount),0)
                from   BTRANSACTION
                where  operation = 'TRANSFER'
                and    Recip = '$sou$';
extract minus = select coalesce(sum(Amount),0)
                from   BTRANSACTION
                where  operation = 'TRANSFER'
                and    Source = '$sou$';
compute balance = $plusD$ + $plusT$ - $minus$;
if ($balance$ < $amnt$);
   showMessage Insufficient balance.@nOperation cancelled.;
   goto END;
endif;
insert into BTRANSACTION
  (TransID,Operation,Source,Recip,Amount,RecTime,Signature)
values ('$tid$','$op$','$acc$','$rec$',$amnt$,'$dat$','$sig$');
```

Script 34.8 - Script executing a transfer between two accounts (excerpts)

This computation could be performed more elegantly with a single SQL query. However, the detailed procedure of Script 34.8 is more intuitive and therefore has

been preferred. It differs from the Bitcoin protocol in that it processes the entire history of the source account since its creation and not only the last transactions the ouptput of which have not yet been consumed. This approach is particularly simple. Though it may seem ineffective, it will be appropriate insofar as:

– the source account history contains a reasonable number of transactions (a few thousands for example),

– indexes can been created on the Source and Recip columns respectively,

– a transfer operation execution time of a one or two seconds will certainly be considered acceptable by the user.

Figure 34.5 represents an example of the contents of a transfer transaction.

TransKey	322
TransID	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=-2019-07-22_13:18:45
Operation	TRANSFER
Source	Vmt2ONz5fHZb8JbpKyzenDLqlLH0pxbxTPeN42Pe2U8=
Recip	g20TQfu12uhOxztWBJ0tIUrNC5L0bhJKMLIUDKTt-Sk=
Amount	18.5
RecDate	2019-07-22 13:18:45
Signature	C75d4HLpm9STHj[..]h-blt2uCmMM_mfK

Figure 34.5 - Composition of a row of table BTRANSACTION describing a TRANSFER operation : *Mary transfers an amount of 18.5 to Luke's account*

The operation of amount transfer is coded in script _BC_Execute_Transfer.sql.

34.11 Validating a transaction

This function checks that the accounts involved in the transaction are recorded in the DIRECTORY table, that the amount of the transfer transaction is legitimate (less than or equal to the balance available on the transaction creation date) and that the values of the components are identical to the content of the signature, decrypted via the public key.

The operation of transaction validation is coded in script _BC_Validate_Transaction.sql.

34.12 Creating and mining a block

This function creates a block containing all the transactions still pending and calculates the nonce that produces a hash value starting with **k** binary zeroes. This hash thus obtained is the block identifier (column Bhash). The identifier of the last block created so far is also added to the new block (column Bprev), therefore increasing the blockchain by one unit.

The procedure of block creation and mining is translated in Script 34.10. It comprises three steps:

1. Building the data of the block

The body of the block is built by concatening the list of pending transactions (those not yet included in a block), the recording time and the hash of the last block of the chain.[4]

The hash of the last block, that will become the previous block, is extracted from the row of the block with the highest recording time[5]:

```
extract previous = select BChash from BLOCK
                   order by RecTime desc;
```

The list of pending transactions is computed in variable transList by a single SQL query that assembles the contents of each transaction according to its type (column Operation) then concatenates them though group_concat aggregate function. For instance, a REGISTER transaction is assembled as follows (as an SQLfast list[6]):

```
TransID||';REGISTER;'||Source||';'||RecTime||';'
       ||Signature
```

In addition, this query counts the transactions to include in the new block (variable blockSize). The body of the block can then be computed:

```
set body = $transList$#$recTime$#$previous$;
```

It is interesting to note that the list of transactions itself is an SQLfast list the separator of which is the **#** character. So, this *list of lists* can be easily disassembled to retrieve the properties of each transaction of the list thanks to the SQLfast list functions. This means that the value of variable body is a valid serialization of the contents of the block that can be broadcasted to the other members of the blockchain through *email*, *p2p* or *ftp* protocols. This distribution aspect is not taken in charge by this prototype.

2. Mining the block

To compute a valid hash and its nonce, we use **miningHash**, a variant of SQLfast function **hash**. This function uses four arguments:

4. Which, actually, is the *building* time, before mining.
5. The `extract` statement processes the first row of the result set. Clause `limit 1` is therefore useless.
6. An SQLfast list is a character string formed by a series of elements separated by a special character (here a semi-colon).

– The character string to hash (in variable body).
– The constraint on the hash, as a **k**-length string of zero characters (in variable prefix); **k** is the *difficulty* of the mining, set to 20 in the standard SQLfast distribution.
– The maximum number of trials (in variable trials); set to 5,000,000 in the standard distribution.
– The format of the resulting hash; 1 for hexadecimal and 2 for base64.

It returns three results:

– hash, the value of a valid hash, if one has been found.
– nonce, the corresponding nonce.
– nHash, the number of hash computations that have been needed to get this valid hash.

If no valid hash has been found, hash = nonce = ''.

```
set k = 20;
compute prefix = repeat('0',$k$);
set trials = 5000000;
function hash,nonce,nHash
         = LStr:miningHash $body$,$prefix$,$trials$,2;
```

The core of this function is shown in Script 34.9. Variable Bdata denotes the string to hash (as a *byte string*, not a *unicode string*) augmented with the current candidate nonce (str(N)). Variable Bhash denotes the sha256 hash of this value, as a binary string. Chash is the prefix of this hash, converted in ASCII characters (one character per bit). Bhash is a valid hash when Chash = prefix.

This procedure certainly is not the fasted we can write, but it is quite sufficient considering the didactic objective of this prototype. A more realistic version would be written in C with true bit manipulation primitives and executed by a cluster of (let us keep it modest) 4,096 processors!

3. Adding the block in the blockchain

A new row is inserted in table BLOCK, comprising the valid hash, the number of transactions, the nonce, the recording time and the hash of the previous block, if any. The rows of the transactions of the block are updated to make them reference the new block.

Figure 34.6 below shows the contents of the rows of the BLOCK table representing blocks with BlockKey 113 and 114 of the chain[7]. We will see how to visualize the transactions of each block in the next section.

7. Prefix 'AAAK' translates '000000 000000 000000 001010' in base64, which complies with *difficulty* k = 20.

```
def miningHash(data,prefix,hashNbr,form):
    lenPrefix = len(prefix)
    nonce = ''
    for N in range(trials):
        Bdata = data.encode('latin-1') + str(N)
        Bhash = hashlib.sha256(Bdata)
        Chash = ''.join('{0:08b}'.format(ord(x), 'b')\
                for x in Bhash.digest()\
                        [:lenPrefix//8+1])[:lenPrefix]
        if Chash == prefix:
            nonce = str(N)
            break
    if form == 2:
        hash = base64.b64encode(Bhash.digest(),'_-')
    return hash,nonce,N
```

Script 34.9 - Main section of Python function LStr:**miningHash** used to mine a block (excerpts)

```
Block n° 114
    Block hash:     AAAKqRd-Zq2PtROoXNLidrG_lYAVT5XnfkKsToEGy1w=
    Nb transact:    5
    Nonce:          1526497
    Creation date:  2019-09-02 12:25:56
    Status:         0
    Previous hash:  AAAKayF8dzDLswTa5rb_V9nP_Gp536vaZDPuInYCIcM=

Block n° 113
    Block hash:     AAAKayF8dzDLswTa5rb_V9nP_Gp536vaZDPuInYCIcM=
    Nb transact:    5
    Nonce:          967544
    Creation date:  2019-09-02 12:15:25
    Status:         0
    Previous hash:  AAAKQ7I-HfRKPSCbFK4OKoJ3vkUdLrC-UXC2GVAtECU=
```

Figure 34.6 - Contents of two successive blocks of 5 transactions

The operation of block creation and mining is coded in script _BC_Create_Block.sql.

34.13 TExamining the blockchain data

Blockchain management systems include an application devoted to the examination of a blockchain and commonly called *Blockchain Explorer*. In this section, we present three exploration functions that can help experimenters understand the result of the blockchain manipulation operations.

```
extract previous = select BChash from BLOCK
                   order by RecTime desc;
extract transList, blockSize
        = select group_concat2(
                  case Operation
                    when 'REGISTER'
                    then TransID||';REGISTER;'||Source
                         ||';'||RecTime||';'||Signature
                    when 'DEPOSIT'  then ...
                    when 'TRANSFER' then ...
                    else 'Transaction type unknown'
                  end,0,TransKey,1,'#'), count(*)
          from   BTRANSACTION
          where  BChash is null;
set recTime = $date$ $clock$;
set data = $transList$#$recTime$#$previous$;
set k = 20;
compute prefix = repeat('0',$k$);
set trials = 5000000;
function hash,nonce,nHash
     = LStr:miningHash $data$,$prefix$,$trials$,2;
insert into BLOCK(BChash,Transact,Nonce,RecTime,BCprev)
       values('$hash$',$blockSize$,'$nonce$','$recTime$',
              case when '$previous$' = '' then null
                   else '$previous$'
              end);
update BTRANSACTION
set    BChash = '$hash$'
where  BChash is null;
```

Script 34.10 - Script creating and mining a block (excerpts)

To illustrate the result produced by these fonctions we suppose that the following operations have been executed, in this order:

1. Mary creates an account
2. Mary deposits an amount of 100 into her account
3. Luke creates an account
4. Mary transfers an amount of 60 to Lucke’s account
5. Luke deposits an amount of 20 into his account
6. Ann creates an account
7. Luke transfers an amount of 35 to Ann's account
8. Luke transfers an amount of 15 to Mary's account
9. Mary transfers an amount of 18 to Ann's account
10. Ann deposits an amount of 30 into her account

11. Ann transfers an amount of 40 to Luke's account
12. Ann transfers an amount of 12 to Mary's account

In addition, a block is created as soon as 5 transactions have been recorded. The final chain will therefore consist of two blocks of 5 transactions and 2 pending transactions.

34.13.1Examining an account

The user selects an account to display the successive transactions of which it is the source or the recipient, as well as its current balance. For reasons of readability, in this function as well as in the other two, accounts are designated by the private name of their owner. The report below is related to Mary's account.

```
Transactions of your account
  +--------+-----------+--------+-------+---------------------+---------+
  | Source | Operation | Amount | Recip | TimeCreated         | InBlock |
  +--------+-----------+--------+-------+---------------------+---------+
  | Mary   | REGISTER  | --     | --    | 2018-12-30 19:25:47 | yes     |
  | --     | DEPOSIT   | 100    | Mary  | 2018-12-30 19:26:09 | yes     |
  | Mary   | TRANSFER  | 60     | Luke  | 2018-12-30 19:26:48 | yes     |
  | Luke   | TRANSFER  | 15     | Mary  | 2018-12-30 19:31:44 | yes     |
  | Mary   | TRANSFER  | 18     | Ann   | 2018-12-30 19:31:59 | yes     |
  | Ann    | TRANSFER  | 12     | Mary  | 2018-12-30 19:36:34 | --      |
  +--------+-----------+--------+-------+---------------------+---------+

Your account balance
  +---------+---------+----------+------------+---------+
  | Account | Deposit | Received | Transfered | Balance |
  +---------+---------+----------+------------+---------+
  | Mary    | 100     | 27       | -78        | 49      |
  +---------+---------+----------+------------+---------+
```

This function is coded in script _BC_Examine_Account.sql.

34.13.2Examining the transactions

The function displays the chronological list of transactions and the balance of each account.

```
Transactions
+--------+-----------+--------+-----------+---------------------+-------+
| Source | Operation | Amount | Recipient | TimeCreated         | Block |
+--------+-----------+--------+-----------+---------------------+-------+
| Mary   | REGISTER  | --     | --        | 2018-12-30 19:25:47 | yes   |
| --     | DEPOSIT   | 100    | Mary      | 2018-12-30 19:26:09 | yes   |
| Luke   | REGISTER  | --     | --        | 2018-12-30 19:26:25 | yes   |
| Mary   | TRANSFER  | 60     | Luke      | 2018-12-30 19:26:48 | yes   |
| --     | DEPOSIT   | 20     | Luke      | 2018-12-30 19:27:02 | yes   |
| Ann    | REGISTER  | --     | --        | 2018-12-30 19:31:00 | yes   |
| Luke   | TRANSFER  | 35     | Ann       | 2018-12-30 19:31:26 | yes   |
| Luke   | TRANSFER  | 15     | Mary      | 2018-12-30 19:31:44 | yes   |
| Mary   | TRANSFER  | 18     | Ann       | 2018-12-30 19:31:59 | yes   |
| --     | DEPOSIT   | 30     | Ann       | 2018-12-30 19:32:12 | yes   |
| Ann    | TRANSFER  | 40     | Luke      | 2018-12-30 19:36:19 | --    |
```

```
| Ann    | TRANSFER  | 12     | Mary      | 2018-12-30 19:36:34 | --    |
+--------+-----------+--------+-----------+---------------------+-------+
```

Account balance

Account	Deposit	Received	Transfered	Balance
Ann	30	53	-52	31
Luke	20	100	-50	70
Mary	100	27	-78	49

This function is coded in script _BC_Examine_Transactions.sql.

34.13.3Examining the blockchain

The function displays the content of each block in the chain, its properties and those of each of its transactions.

Block n° 2

```
      Block hash:    AAALlzLVgFBSa_E5Wzqf1RQ6VlK8M1mVkrlvHha26r4=
      Nbr transact:  5
      Nonce:         334045
      Record Time:   2018-12-30 19:35:05
      Status:        0
      Hash previous: AAAD8ewt9-jHhBJts048MamR-vbeCts4BXLCpZh8QsY=
```

Source	Operation	Amount	Recipient	TimeCreated
Ann	REGISTER	--	--	2018-12-30 19:31:00
Luke	TRANSFER	35	Ann	2018-12-30 19:31:26
Luke	TRANSFER	15	Mary	2018-12-30 19:31:44
Mary	TRANSFER	18	Ann	2018-12-30 19:31:59
--	DEPOSIT	30	Ann	2018-12-30 19:32:12

Block n° 1

```
      Block hash:    AAAD8ewt9-jHhBJts048MamR-vbeCts4BXLCpZh8QsY=
      Nbr transact:  5
      Nonce:         1815939
      Record Time:   2018-12-30 19:27:10
      Status:        0
      Hash previous: --
```

Source	Operation	Amount	Recipient	TimeCreated
Mary	REGISTER	--	--	2018-12-30 19:25:47
--	DEPOSIT	100	Mary	2018-12-30 19:26:09
Luke	REGISTER	--	--	2018-12-30 19:26:25
Mary	TRANSFER	60	Luke	2018-12-30 19:26:48
--	DEPOSIT	20	Luke	2018-12-30 19:27:02

Pending transactions (not yet in block)

Source	Operation	Amount	Recipient	TimeCreated
Ann	TRANSFER	40	Luke	2018-12-30 19:36:19
Ann	TRANSFER	12	Mary	2018-12-30 19:36:34

This function is coded in script _BC_Examine_Blockchain.sql.

34.14 Wrapping it all: the BLOCKCHAIN toolbox

The functions of the toolbox are independent, but, for convenience, they are available via the control screen in Figure 34.7, displayed by script **BLOCKCHAIN.sql**. The execution of the three operations generating a transaction can be accompanied by the display of the contents of the ACCOUNT, DIRECTORY and BTRANSACTION tables. To do this, check the box entitled "Display table contents after each transaction".

The table contents can also be viewed via the "Database > Show DB Data" menu function in the main SQLfast window.

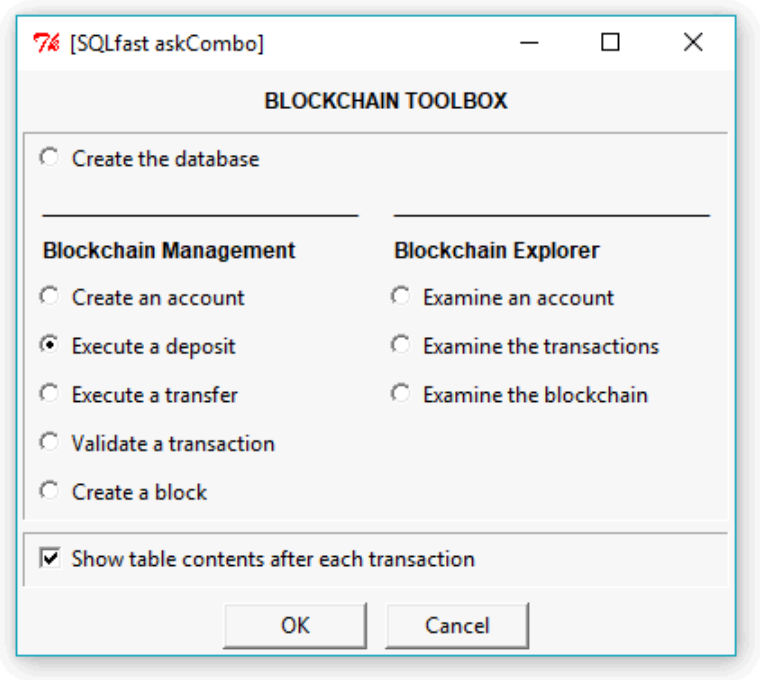

Figure 34.7 - Control panel of the toolbox

The scripts of the prototype Blockchain toolbox are available in the SQLfast/Scripts/Case-Studies/Case_BlockChain directory.