BigchainDB Documentation

BigchainDB Contributors

Aug 13, 2018

Contents

1 More About BigchainDB 3

Meet BigchainDB. The blockchain database.

It has some database characteristics and some blockchain characteristics, including decentralization, immutability and native support for assets.

At a high level, one can communicate with a BigchainDB cluster (set of nodes) using the BigchainDB HTTP API, or a wrapper for that API, such as the BigchainDB Python Driver. Each BigchainDB node runs BigchainDB Server and various other software. The terminology page explains some of those terms in more detail.

CHAPTER 1

More About BigchainDB

1.1 Production-Ready?

Depending on your use case, BigchainDB may or may not be production-ready. You should ask your service provider. If you want to go live (into production) with BigchainDB, please consult with your service provider.

Note: BigchainDB has an open source license with a "no warranty" section that is typical of open source licenses. This is standard in the software industry. For example, the Linux kernel is used in production by billions of machines even though its license includes a "no warranty" section. Warranties are usually provided above the level of the software license, by service providers.

1.2 Terminology

There is some specialized terminology associated with BigchainDB. To get started, you should at least know the following:

1.2.1 BigchainDB Node

A **BigchainDB node** is a machine (or logical machine) running BigchainDB Server and related software. Each node is controlled by one person or organization.

1.2.2 BigchainDB Cluster

A set of BigchainDB nodes can connect to each other to form a **BigchainDB cluster**. Each node in the cluster runs the same software. A cluster may have additional machines to do things such as cluster monitoring.

1.2.3 BigchainDB Consortium

The people and organizations that run the nodes in a cluster belong to a **BigchainDB consortium** (i.e. another organization). A consortium must have some sort of governance structure to make decisions. If a cluster is run by a single company, then the "consortium" is just that company.

What's the Difference Between a Cluster and a Consortium?

A cluster is just a bunch of connected nodes. A consortium is an organization which has a cluster, and where each node in the cluster has a different operator.

1.3 How BigchainDB is Decentralized

Decentralization means that no one owns or controls everything, and there is no single point of failure.

Ideally, each node in a BigchainDB cluster is owned and controlled by a different person or organization. Even if the cluster lives within one organization, it's still preferable to have each node controlled by a different person or subdivision.

We use the phrase "BigchainDB consortium" (or just "consortium") to refer to the set of people and/or organizations who run the nodes of a BigchainDB cluster. A consortium requires some form of governance to make decisions such as membership and policies. The exact details of the governance process are determined by each consortium, but it can be very decentralized.

If sharding is turned on (i.e. if the number of shards is larger than one), then the actual data is decentralized in that no one node stores all the data.

Every node has its own locally-stored list of the public keys of other consortium members: the so-called keyring. There's no centrally-stored or centrally-shared keyring.

A consortium can increase its decentralization (and its resilience) by increasing its jurisdictional diversity, geographic diversity, and other kinds of diversity. This idea is expanded upon in the section on node diversity.

There's no node that has a long-term special position in the cluster. All nodes run the same software and perform the same duties.

If someone has (or gets) admin access to a node, they can mess with that node (e.g. change or delete data stored on that node), but those changes should remain isolated to that node. The BigchainDB cluster can only be compromised if more than one third of the nodes get compromised. See the Tendermint documentation for more details.

It's worth noting that not even the admin or superuser of a node can transfer assets. The only way to create a valid transfer transaction is to fulfill the current crypto-conditions on the asset, and the admin/superuser can't do that because the admin user doesn't have the necessary information (e.g. private keys).

1.4 Kinds of Node Diversity

Steps should be taken to make it difficult for any one actor or event to control or damage "enough" of the nodes. (Because BigchainDB Server uses Tendermint, "enough" is .) There are many kinds of diversity to consider, listed below. It may be quite difficult to have high diversity of all kinds.

1. **Jurisdictional diversity.** The nodes should be controlled by entities within multiple legal jurisdictions, so that it becomes difficult to use legal means to compel enough of them to do something.
2. **Geographic diversity.** The servers should be physically located at multiple geographic locations, so that it becomes difficult for a natural disaster (such as a flood or earthquake) to damage enough of them to cause problems.

3. **Hosting diversity.** The servers should be hosted by multiple hosting providers (e.g. Amazon Web Services, Microsoft Azure, Digital Ocean, Rackspace), so that it becomes difficult for one hosting provider to influence enough of the nodes.
4. **Diversity in general.** In general, membership diversity (of all kinds) confers many advantages on a consortium. For example, it provides the consortium with a source of various ideas for addressing challenges.

Note: If all the nodes are running the same code, i.e. the same implementation of BigchainDB, then a bug in that code could be used to compromise all of the nodes. Ideally, there would be several different, well-maintained implementations of BigchainDB Server (e.g. one in Python, one in Go, etc.), so that a consortium could also have a diversity of server implementations. Similar remarks can be made about the operating system.

1.5 How BigchainDB is Immutable

The word *immutable* means "unchanging over time or unable to be changed." For example, the decimal digits of π are immutable (3.14159...).

The blockchain community often describes blockchains as "immutable." If we interpret that word literally, it means that blockchain data is unchangeable or permanent, which is absurd. The data *can* be changed. For example, a plague might drive humanity extinct; the data would then get corrupted over time due to water damage, thermal noise, and the general increase of entropy.

It's true that blockchain data is more difficult to change (or delete) than usual. It's more than just "tamper-resistant" (which implies intent), blockchain data also resists random changes that can happen without any intent, such as data corruption on a hard drive. Therefore, in the context of blockchains, we interpret the word "immutable" to mean *practically* immutable, for all intents and purposes. (Linguists would say that the word "immutable" is a *term of art* in the blockchain community.)

Blockchain data can be made immutable in several ways:

1. **No APIs for changing or deleting data.** Blockchain software usually doesn't expose any APIs for changing or deleting the data stored in the blockchain. BigchainDB has no such APIs. This doesn't prevent changes or deletions from happening in *other* ways; it's just one line of defense.
2. **Replication.** All data is replicated (copied) to several different places. The higher the replication factor, the more difficult it becomes to change or delete all replicas.
3. **Internal watchdogs.** All nodes monitor all changes and if some unallowed change happens, then appropriate action can be taken.
4. **External watchdogs.** A consortium may opt to have trusted third-parties to monitor and audit their data, looking for irregularities. For a consortium with publicly-readable data, the public can act as an auditor.
5. **Economic incentives.** Some blockchain systems make it very expensive to change old stored data. Examples include proof-of-work and proof-of-stake systems. BigchainDB doesn't use explicit incentives like those.
6. Data can be stored using fancy techniques, such as error-correction codes, to make some kinds of changes easier to undo.
7. **Cryptographic signatures** are often used as a way to check if messages (e.g. transactions) have been tampered with enroute, and as a way to verify who signed the messages. In BigchainDB, each transaction must be signed by one or more parties.
8. **Full or partial backups** may be recorded from time to time, possibly on magnetic tape storage, other blockchains, printouts, etc.
9. **Strong security.** Node owners can adopt and enforce strong security policies.
10. **Node diversity.** Diversity makes it so that no one thing (e.g. natural disaster or operating system bug) can compromise enough of the nodes. See the section on the kinds of node diversity.

1.6 BigchainDB and Byzantine Fault Tolerance

BigchainDB Server uses Tendermint for consensus and transaction replication, and Tendermint is Byzantine Fault Tolerant (BFT).

1.7 Querying BigchainDB

A node operator can use the full power of MongoDB's query engine to search and query all stored data, including all transactions, assets and metadata. The node operator can decide for themselves how much of that query power they expose to external users.

1.7.1 Blog Post with Example Queries

We wrote a blog post in The BigchainDB Blog to show how to use some MongoDB tools to query a BigchainDB node's MongoDB database. It includes some specific example queries for data about custom cars and their ownership histories. Check it out.

1.7.2 How to Connect to MongoDB

Before you can query a MongoDB database, you must connect to it, and to do that, you need to know its hostname and port.

If you're running a BigchainDB node on your local machine (e.g. for dev and test), then the hostname should be `localhost` and the port should be `27017`, unless you did something to change those values. If you're running a BigchainDB node on a remote machine and you can SSH to that machine, then the same is true.

If you're running a BigchainDB node on a remote machine and you configured its MongoDB to use auth and to be publicly-accessible (to people with authorization), then you can probably figure out its hostname and port.

1.7.3 How to Query

A BigchainDB node operator has full access to their local MongoDB instance, so they can use any of MongoDB's APIs for running queries, including:

- the Mongo Shell,
- MongoDB Compass,
- one of the MongoDB drivers, such as PyMongo, or
- a third-party tool for doing MongoDB queries, such as RazorSQL, Studio 3T, Mongo Management Studio, NoSQLBooster for MongoDB, or Dr. Mongo.

Note: It's possible to do query a MongoDB database using SQL. For example:

- Studio 3T: "How to Query MongoDB with SQL"
- NoSQLBooster for MongoDB: "How to Query MongoDB with SQL SELECT"

For example, if you're on a machine that's running a default BigchainDB node, then you can connect to it using the Mongo Shell (`mongo`) and look around like so:

```
$ mongo
MongoDB shell version v3.6.5
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.4
...
> show dbs
admin     0.000GB
bigchain  0.000GB
config    0.000GB
local     0.000GB
> use bigchain
switched to db bigchain
> show collections
assets
blocks
metadata
pre_commit
transactions
utxos
validators
```

The above example illustrates several things:

- When you don't specify the hostname or port, the Mongo Shell assumes they are `localhost` and `27017`, respectively. (`localhost` had IP address 127.0.0.1 on the machine in question, an Ubuntu machine.)
- BigchainDB stores its data in a database named `bigchain`.
- The `bigchain` database contains several collections.
- Votes aren't stored in any collection, currently. They are all handled and stored by Tendermint in its own (LevelDB) database.

1.7.4 Example Documents from Some Collections

The most interesting collections in the `bigchain` database are:

- transactions
- assets
- metadata
- blocks

You can explore those collections using MongoDB queries such as `db.assets.findOne()`. We now show some example documents from each of those collections.

Example Documents from transactions

A CREATE transaction from the transactions collection includes an extra `"_id"` field (added by MongoDB) and is missing its `"asset"` and `"metadata"` fields: that data was removed and stored in the assets and metadata collections.

```
{
    "_id":ObjectId("5b17b9fa6ce88300067b6804"),
    "inputs":[...],
    "outputs":[...],
```

(continues on next page)

(continued from previous page)

```
    "operation":"CREATE",
    "version":"2.0",
    "id":"816c4dd7...851af1629"
}
```

A TRANSFER transaction from the transactions collection is similar, but it keeps its `"asset"` field.

```
{
    "_id":ObjectId("5b17b9fa6ce88300067b6807"),
    "inputs":[...],
    "outputs":[...],
    "operation":"TRANSFER",
    "asset":{
        "id":"816c4dd7ae...51af1629"
    },
    "version":"2.0",
    "id":"985ee697d...a3296b9"
}
```

Example Document from assets

A document from the assets collection has three top-level fields: an `"_id"` field added by MongoDB, the `asset.data` from a CREATE transaction, and the `"id"` of the CREATE transaction it came from.

```
{
    "_id":ObjectId("5b17b9fe6ce88300067b6823"),
    "data":{
        "type":"cow",
        "name":"Mildred"
    },
    "id":"96002ef8740...45869959d8"
}
```

Example Document from metadata

A document from the metadata collection has three top-level fields: an `"_id"` field added by MongoDB, the `metadata` from a transaction, and the `"id"` of the transaction it came from.

```
{
    "_id":ObjectId("5b17ba006ce88300067b683d"),
    "metadata":{
        "transfer_time":1058568256
    },
    "id":"53cba620e...ae9fdee0"
}
```

Example Document from blocks

```
{
    "_id":ObjectId("5b212c1ceaaa420006f41c57"),
    "app_hash":"2b0b75c2c2...7fb2652ce26c6",
```

(continues on next page)

(continued from previous page)

```
    "height":17,
    "transactions":[
        "5f1f2d6b...ed98c1e"
    ]
}
```

1.7.5 What a Node Operator Can Expose to External Users

Each node operator can decide how they let external users get information from their local MongoDB database. They could expose:

- their local MonogoDB database itself to queries from external users, maybe as a MongoDB user with a role that has limited privileges, e.g. read-only.
- a limited HTTP API, allowing a restricted set of predefined queries, such as the HTTP API provided by BigchainDB Server, or a custom HTTP API implemented using Django, Express, Ruby on Rails, or ASP.NET.
- some other API, such as a GraphQL API. They could do that using custom code or code from a third party.

Each node operator can expose a different level or type of access to their local MongoDB database. For example, one node operator might decide to specialize in offering optimized geospatial queries.

1.7.6 Security Considerations

In BigchainDB version 1.3.0 and earlier, there was one logical MongoDB database, so exposing that database to external users was very risky, and was not recommended. "Drop database" would delete that one shared MongoDB database.

In BigchainDB version 2.0.0 and later, each node has its own isolated local MongoDB database. Inter-node communications are done using Tendermint protocols, not MongoDB protocols, as illustrated in Figure 1 below. If a node's local MongoDB database gets compromised, none of the other MongoDB databases (in the other nodes) will be affected.

1.7.7 Performance and Cost Considerations

Query processing can be quite resource-intensive, so it's a good idea to have MongoDB running in a separate machine from those running BigchainDB Server and Tendermint Core.

A node operator might want to measure the resources used by a query, so they can charge whoever requested the query accordingly.

Some queries can take too long or use too many resources. A node operator should put upper bounds on the resources that a query can use, and halt (or prevent) any query that goes over.

To make MongoDB queries more efficient, one can create indexes. Those indexes might be created by the node operator or by some external users (if the node operator allows that). It's worth noting that indexes aren't free: whenever new data is appended to a collection, the corresponding indexes must be updated. The node operator might want to pass those costs on to whoever created the index. Moreover, in MongoDB, a single collection can have no more than 64 indexes.

One can create a follower node: a node with Tendermint voting power 0. It would still have a copy of all the data, so it could be used as read-only node. A follower node could offer specialized queries as a service without affecting the workload on the voting validators (which can also write). There could even be followers of followers.

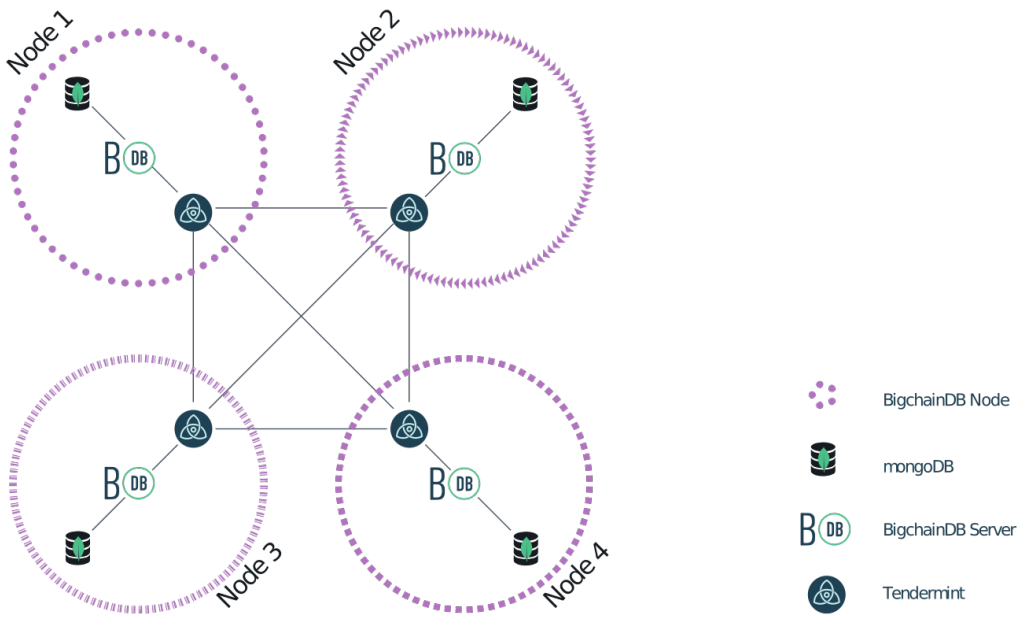


Fig. 1: Figure 1: A Four-Node BigchainDB 2.0 Network

1.7.8 JavaScript Query Code Examples

One can connect to a node's MongoDB database using any of the MongoDB drivers, such as the MongoDB Node.js driver. Here are some links to example JavaScript code that queries a BigchainDB node's MongoDB database:

- The BigchainDB JavaScript/Node.js driver source code
- Example code by @manolodewiner
- More example code by @manolodewiner

1.8 How BigchainDB is Good for Asset Registrations & Transfers

BigchainDB can store data of any kind (within reason), but it's designed to be particularly good for storing asset registrations and transfers:

- The fundamental thing that one sends to a BigchainDB cluster, to be checked and stored (if valid), is a *transaction*, and there are two kinds: CREATE transactions and TRANSFER transactions.
- A CREATE transaction can be use to register any kind of asset (divisible or indivisible), along with arbitrary metadata.
- An asset can have zero, one, or several owners.
- The owners of an asset can specify (crypto-)conditions which must be satisfied by anyone wishing transfer the asset to new owners. For example, a condition might be that at least 3 of the 5 current owners must cryptographically sign a TRANSFER transaction.
- BigchainDB verifies that the conditions have been satisfied as part of checking the validity of TRANSFER transactions. (Moreover, anyone can check that they were satisfied.)
- BigchainDB prevents double-spending of an asset.
- Validated transactions are *"immutable"*.

Note: We used the word "owners" somewhat loosely above. A more accurate word might be fulfillers, signers, controllers, or transfer-enablers. See the section titled **A Note about Owners** in the relevant BigchainDB Transactions Spec.

1.9 BigchainDB and Smart Contracts

One can store the source code of any smart contract (i.e. a computer program) in BigchainDB, but BigchainDB won't run arbitrary smart contracts.

BigchainDB will run the subset of smart contracts expressible using Crypto-Conditions.

The owners of an asset can impose conditions on it that must be met for the asset to be transferred to new owners. Examples of possible conditions (crypto-conditions) include:

- The current owner must sign the transfer transaction (one which transfers ownership to new owners).
- Three out of five current owners must sign the transfer transaction.
- (Shannon and Kelly) or Morgan must sign the transfer transaction.

Crypto-conditions can be quite complex. They can't include loops or recursion and therefore will always run/check in finite time.

Note: We used the word "owners" somewhat loosely above. A more accurate word might be fulfillers, signers, controllers, or transfer-enablers. See the section titled **A Note about Owners** in the relevant BigchainDB Transactions Spec.

1.10 Transaction Concepts

In BigchainDB, *transactions* are used to register, issue, create or transfer things (e.g. assets).

Transactions are the most basic kind of record stored by BigchainDB. There are two kinds: CREATE transactions and TRANSFER transactions.

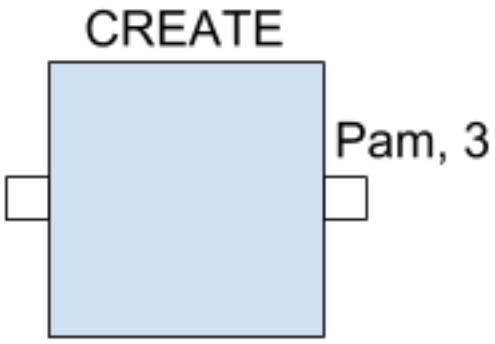
1.10.1 CREATE Transactions

A CREATE transaction can be used to register, issue, create or otherwise initiate the history of a single thing (or asset) in BigchainDB. For example, one might register an identity or a creative work. The things are often called "assets" but they might not be literal assets.

BigchainDB supports divisible assets as of BigchainDB Server v0.8.0. That means you can create/register an asset with an initial number of "shares." For example, A CREATE transaction could register a truckload of 50 oak trees. Each share of a divisible asset must be interchangeable with each other share; the shares must be fungible.

A CREATE transaction can have one or more outputs. Each output has an associated amount: the number of shares tied to that output. For example, if the asset consists of 50 oak trees, one output might have 35 oak trees for one set of owners, and the other output might have 15 oak trees for another set of owners.

Each output also has an associated condition: the condition that must be met (by a TRANSFER transaction) to transfer/spend the output. BigchainDB supports a variety of conditions. For details, see the section titled **Transaction Components: Conditions** in the relevant BigchainDB Transactions Spec.



Above we see a diagram of an example BigchainDB CREATE transaction. It has one output: Pam owns/controls three shares of the asset and there are no other shares (because there are no other outputs).

Each output also has a list of all the public keys associated with the conditions on that output. Loosely speaking, that list might be interpreted as the list of "owners." A more accurate word might be fulfillers, signers, controllers, or transfer-enablers. See the section titled **A Note about Owners** in the relevant BigchainDB Transactions Spec.

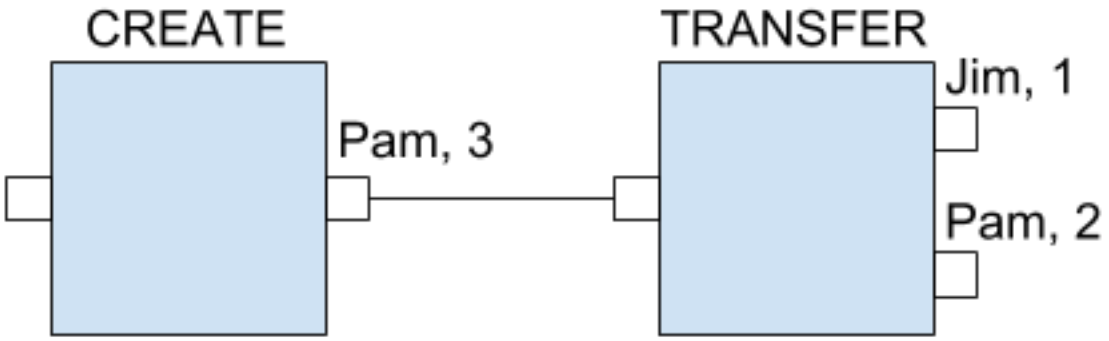
A CREATE transaction must be signed by all the owners. (If you're looking for that signature, it's in the one "fulfillment" of the one input, albeit encoded.)

1.10.2 TRANSFER Transactions

A TRANSFER transaction can transfer/spend one or more outputs on other transactions (CREATE transactions or other TRANSFER transactions). Those outputs must all be associated with the same asset; a TRANSFER transaction can only transfer shares of one asset at a time.

Each input on a TRANSFER transaction connects to one output on another transaction. Each input must satisfy the condition on the output it's trying to transfer/spend.

A TRANSFER transaction can have one or more outputs, just like a CREATE transaction (described above). The total number of shares coming in on the inputs must equal the total number of shares going out on the outputs.



Above we see a diagram of two example BigchainDB transactions, a CREATE transaction and a TRANSFER transaction. The CREATE transaction is the same as in the earlier diagram. The TRANSFER transaction spends Pam's output, so the input on that TRANSFER transaction must contain a valid signature from Pam (i.e. a valid fulfillment). The TRANSFER transaction has two outputs: Jim gets one share, and Pam gets the remaining two shares.

Terminology: The "Pam, 3" output is called a "spent transaction output" and the "Jim, 1" and "Pam, 2" outputs are called "unspent transaction outputs" (UTXOs).

Example 1: Suppose a red car is owned and controlled by Joe. Suppose the current transfer condition on the car says that any valid transfer must be signed by Joe. Joe could build a TRANSFER transaction containing an input with Joe's signature (to fulfill the current output condition) plus a new output condition saying that any valid transfer must be signed by Rae.

Example 2: Someone might construct a TRANSFER transaction that fulfills the output conditions on four previously-untransferred assets of the same asset type e.g. paperclips. The amounts might be 20, 10, 45 and 25, say, for a total of 100 paperclips. The TRANSFER transaction would also set up new transfer conditions. For example, maybe a set of 60 paperclips can only be transferred if Gertrude signs, and a separate set of 40 paperclips can only be transferred if both Jack and Kelly sign. Note how the sum of the incoming paperclips must equal the sum of the outgoing paperclips (100).

1.10.3 Transaction Validity

When a node is asked to check if a transaction is valid, it checks several things. We documented those things in a post on *The BigchainDB Blog*: "What is a Valid Transaction in BigchainDB?" (Note: That post was about BigchainDB Server v1.0.0.)

Each BigchainDB Transactions Spec documents the conditions for a transaction (of that version) to be valid.

1.10.4 Example Transactions

There are example BigchainDB transactions in the HTTP API documentation and the Python Driver documentation.

1.11 How to Store Files in BigchainDB

While it's possible to store a file in a BigchainDB network, we don't recommend doing that. It works best for storing, indexing and querying *structured data*, not files.

If you want decentralized file storage, check out Storj, Sia, Swarm or IPFS/Filecoin. You could store file URLs, hashes or other metadata in a BigchainDB network.

If you really must store a file in a BigchainDB network, then one way to do that is to convert it to a long Base64 string and then to store that string in one or more BigchainDB transactions, either in the `asset.data` of a CREATE transaction, or the `metadata` of any transaction.

1.12 Permissions in BigchainDB

BigchainDB lets users control what other users can do, to some extent. That ability resembles "permissions" in the *nix world, "privileges" in the SQL world, and "access control" in the security world.

1.12.1 Permission to Spend/Transfer an Output

In BigchainDB, every output has an associated condition (crypto-condition).

To spend/transfer an unspent output, a user (or group of users) must fulfill the condition. Another way to say that is that only certain users have permission to spend the output. The simplest condition is of the form, "Only someone with the private key corresponding to this public key can spend this output." Much more elaborate conditions are possible, e.g. "To spend this output, . . . "

- ". . . anyone in the Accounting Group can sign."
- ". . . three of these four people must sign."
- ". . . either Bob must sign, or both Tom and Sylvia must sign."

For details, see the section titled **Transaction Components: Conditions** in the relevant BigchainDB Transactions Spec.

Once an output has been spent, it can't be spent again: *nobody* has permission to do that. That is, BigchainDB doesn't permit anyone to "double spend" an output.

1.12.2 Write Permissions

When someone builds a TRANSFER transaction, they can put an arbitrary JSON object in the `metadata` field (within reason; real BigchainDB networks put a limit on the size of transactions). That is, they can write just about anything they want in a TRANSFER transaction.

Does that mean there are no "write permissions" in BigchainDB? Not at all!

A TRANSFER transaction will only be valid (allowed) if its inputs fulfill some previous outputs. The conditions on those outputs will control who can build valid TRANSFER transactions. In other words, one can interpret the condition on an output as giving "write permissions" to certain users to write something into the history of the associated asset.

As a concrete example, you could use BigchainDB to write a public journal where only you have write permissions. Here's how: First you'd build a CREATE transaction with the `asset.data` being something like `{"title": "The Journal of John Doe"}`, with one output. That output would have an amount 1 and a condition that only you (who has your private key) can spend that output. Each time you want to append something to your journal, you'd build a new TRANSFER transaction with your latest entry in the `metadata` field, e.g.

```
{"timestamp": "1508319582",
 "entry": "I visited Marmot Lake with Jane."}
```

The TRANSFER transaction would have one output. That output would have an amount 1 and a condition that only you (who has your private key) can spend that output. And so on. Only you would be able to append to the history of that asset (your journal).

The same technique could be used for scientific notebooks, supply-chain records, government meeting minutes, and so on.

You could do more elaborate things too. As one example, each time someone writes a TRANSFER transaction, they give *someone else* permission to spend it, setting up a sort of writers-relay or chain letter.

Note: Anyone can write any JSON (again, within reason) in the `asset.data` field of a CREATE transaction. They don't need permission.

1.12.3 Read Permissions

See the page titled, *BigchainDB, Privacy and Private Data*.

1.12.4 Role-Based Access Control (RBAC)

In September 2017, we published a blog post about how one can define an RBAC sub-system on top of BigchainDB. At the time of writing (January 2018), doing so required the use of a plugin, so it's not possible using standard BigchainDB (which is what's available on the BigchainDB Testnet). That may change in the future. If you're interested, contact BigchainDB.

1.13 BigchainDB, Privacy and Private Data

1.13.1 Basic Facts

1. One can store arbitrary data (including encrypted data) in a BigchainDB network, within limits: there's a maximum transaction size. Every transaction has a `metadata` section which can store almost any Unicode string (up to some maximum length). Similarly, every CREATE transaction has an `asset.data` section which can store almost any Unicode string.
2. The data stored in certain BigchainDB transaction fields must not be encrypted, e.g. public keys and amounts. BigchainDB doesn't offer private transactions akin to Zcoin.
3. Once data has been stored in a BigchainDB network, it's best to assume it can't be change or deleted.
4. Every node in a BigchainDB network has a full copy of all the stored data.

5. Every node in a BigchainDB network can read all the stored data.
6. Everyone with full access to a BigchainDB node (e.g. the sysadmin of a node) can read all the data stored on that node.
7. Everyone given access to a node via the BigchainDB HTTP API can find and read all the data stored by BigchainDB. The list of people with access might be quite short.
8. If the connection between an external user and a BigchainDB node isn't encrypted (using HTTPS, for example), then a wiretapper can read all HTTP requests and responses in transit.
9. If someone gets access to plaintext (regardless of where they got it), then they can (in principle) share it with the whole world. One can make it difficult for them to do that, e.g. if it is a lot of data and they only get access inside a secure room where they are searched as they leave the room.

1.13.2 Storing Private Data Off-Chain

A system could store data off-chain, e.g. in a third-party database, document store, or content management system (CMS) and it could use BigchainDB to:

- Keep track of who has read permissions (or other permissions) in a third-party system. An example of how this could be done is described below.
- Keep a permanent record of all requests made to the third-party system.
- Store hashes of documents-stored-elsewhere, so that a change in any document can be detected.
- Record all handshake-establishing requests and responses between two off-chain parties (e.g. a Diffie-Hellman key exchange), so as to prove that they established an encrypted tunnel (without giving readers access to that tunnel). There are more details about this idea in the BigchainDB Privacy Protocols repository.

A simple way to record who has read permission on a particular document would be for the third-party system ("DocPile") to store a CREATE transaction in a BigchainDB network for every document+user pair, to indicate that that user has read permissions for that document. The transaction could be signed by DocPile (or maybe by a document owner, as a variation). The asset data field would contain 1) the unique ID of the user and 2) the unique ID of the document. The one output on the CREATE transaction would only be transferable/spendable by DocPile (or, again, a document owner).

To revoke the read permission, DocPile could create a TRANSFER transaction, to spend the one output on the original CREATE transaction, with a metadata field to say that the user in question no longer has read permission on that document.

This can be carried on indefinitely, i.e. another TRANSFER transaction could be created by DocPile to indicate that the user now has read permissions again.

DocPile can figure out if a given user has read permissions on a given document by reading the last transaction in the CREATE $\rightarrow$ TRANSFER $\rightarrow$ TRANSFER $\rightarrow$ etc. chain for that user+document pair.

There are other ways to accomplish the same thing. The above is just one example.

You might have noticed that the above example didn't treat the "read permission" as an asset owned (controlled) by a user because if the permission asset is given to (transferred to or created by) the user then it cannot be controlled any further (by DocPile) until the user transfers it back to DocPile. Moreover, the user could transfer the asset to someone else, which might be problematic.

1.13.3 Storing Private Data On-Chain, Encrypted

There are many ways to store private data on-chain, encrypted. Every use case has its own objectives and constraints, and the best solution depends on the use case. The BigchainDB consulting team, along with our partners, can help

you design the best solution for your use case.

Below we describe some example system setups, using various crypto primitives, to give a sense of what's possible.

Please note:

- Ed25519 keypairs are designed for signing and verifying cryptographic signatures, not for encrypting and decrypting messages. For encryption, you should use keypairs designed for encryption, such as X25519.
- If someone (or some group) publishes how to decrypt some encrypted data on-chain, then anyone with access to that encrypted data will be able to get the plaintext. The data can't be deleted.
- Encrypted data can't be indexed or searched by MongoDB. (It can index and search the ciphertext, but that's not very useful.) One might use homomorphic encryption to index and search encrypted data, but MongoDB doesn't have any plans to support that any time soon. If there is indexing or keyword search needed, then some fields of the `asset.data` or `metadata` objects can be left as plain text and the sensitive information can be stored in an encrypted child-object.

System Example 1

Encrypt the data with a symmetric key and store the ciphertext on-chain (in `metadata` or `asset.data`). To communicate the key to a third party, use their public key to encrypt the symmetric key and send them that. They can decrypt the symmetric key with their private key, and then use that symmetric key to decrypt the on-chain ciphertext.

The reason for using a symmetric key along with public/private keypairs is so the ciphertext only has to be stored once.

System Example 2

This example uses proxy re-encryption:

1. MegaCorp encrypts some data using its own public key, then stores that encrypted data (ciphertext 1) in a BigchainDB network.
2. MegaCorp wants to let others read that encrypted data, but without ever sharing their private key and without having to re-encrypt themselves for every new recipient. Instead, they find a "proxy" named Moxie, to provide proxy re-encryption services.
3. Zorban contacts MegaCorp and asks for permission to read the data.
4. MegaCorp asks Zorban for his public key.
5. MegaCorp generates a "re-encryption key" and sends it to their proxy, Moxie.
6. Moxie (the proxy) uses the re-encryption key to encrypt ciphertext 1, creating ciphertext 2.
7. Moxie sends ciphertext 2 to Zorban (or to MegaCorp who forwards it to Zorban).
8. Zorban uses his private key to decrypt ciphertext 2, getting the original un-encrypted data.

Note:

- The proxy only ever sees ciphertext. They never see any un-encrypted data.
- Zorban never got the ability to decrypt ciphertext 1, i.e. the on-chain data.
- There are variations on the above flow.

System Example 3

This example uses erasure coding:

1. Erasure-code the data into n pieces.
2. Encrypt each of the n pieces with a different encryption key.
3. Store the n encrypted pieces on-chain, e.g. in n separate transactions.
4. Share each of the the n decryption keys with a different party.

If k < N of the key-holders gets and decrypts k of the pieces, they can reconstruct the original plaintext. Less than k would not be enough.

System Example 4

This setup could be used in an enterprise blockchain scenario where a special node should be able to see parts of the data, but the others should not.

- The special node generates an X25519 keypair (or similar asymmetric *encryption* keypair).
- A BigchainDB end user finds out the X25519 public key (encryption key) of the special node.
- The end user creates a valid BigchainDB transaction, with either the asset.data or the metadata (or both) encrypted using the above-mentioned public key.
- This is only done for transactions where the contents of asset.data or metadata don't matter for validation, so all node operators can validate the transaction.
- The special node is able to decrypt the encrypted data, but the other node operators can't, and nor can any other end user.