BigchainDB Documentation

Release

BigchainDB Contributors

Apr 24, 2018

Contents

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.

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.

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?

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. In the case of Bitcoin, nothing so drastic is required: a 51% attack will suffice.

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 achieve immutability in several ways:

1. **Replication.** All data is replicated (copied) to several different places. The replication factor can be set by the consortium. The higher the replication factor, the more difficult it becomes to change or delete all replicas.

2. **Internal watchdogs.** All nodes monitor all changes and if some unallowed change happens, then appropriate action can be taken.

3. **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.

4. **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.

5. Data can be stored using fancy techniques, such as error-correction codes, to make some kinds of changes easier to undo.

6. **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.

7. **Full or partial backups** may be recorded from time to time, possibly on magnetic tape storage, other blockchains, printouts, etc.

8. **Strong security.** Node owners can adopt and enforce strong security policies.

9. **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.

Some of these things come "for free" as part of the BigchainDB software, and others require some extra effort from the consortium and node owners.

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 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,

- one of the MongoDB drivers, such as PyMongo, or

- a third-party tool or driver for doing MongoDB queries, such as RazorSQL.

1.7.2 What Can be Queried?

BigchainDB Server creates several MongoDB collections in the node's local MongoDB database. You can see the list of collections by looking at the `create_tables` method in the BigchainDB Server file `bigchaindb/backend/localmongodb/schema.py`. The most interesting collections are:

- transactions

- assets

- metadata

- blocks

We don't detail what's in each collection here, but the collection names are fairly self-explanatory. You can explore their contents using MongoDB queries. A couple of things worth noting are:

1. The transactions collection doesn't include any `asset.data` or `metadata` values (JSON documents). Those are all removed and stored separately in the assets and metadata collections, respectively.

2. The JSON documents stored in the blocks collection are *not* Tendermint blocks, they are BigchainDB blocks.

3. Votes aren't stored in any MongoDB collection, currently. They are all handled and stored by Tendermint in its own (LevelDB) database.

1.7.3 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 MongoDB 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.

1.7.4 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.

Fig. 1.1: Figure 1: A Four-Node BigchainDB 2.0 Network

1.7.5 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.

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 note about "owners" in the IPDB Transaction 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. Crypto-conditions are part of the Interledger Protocol.

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 note about "owners" in the IPDB Transaction 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, a subset of the Interledger Protocol (ILP) crypto-conditions. For details, see the section about conditions in the IPDB Transaction Spec.

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 note about "owners" in the IPDB Transaction 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.

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.)

The IPDB Transaction Spec documents the conditions for a transaction 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 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.

1.11.1 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 about conditions in the IPDB Transaction 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.

1.11.2 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.

1.11.3 Read Permissions

All the data stored in a BigchainDB network can be read by anyone with access to that network. One *can* store encrypted data, but if the decryption key ever leaks out, then the encrypted data can be read, decrypted, and leak out too. (Deleting the encrypted data is *not an option*.)

The permission to read some specific information (e.g. a music file) can be thought of as an *asset*. (In many countries, that permission or "right" is a kind of intellectual property.) BigchainDB can be used to register that asset and transfer it from owner to owner. Today, BigchainDB does not have a way to restrict read access of data stored in a BigchainDB network, but many third-party services do offer that (e.g. Google Docs, Dropbox). In principle, a third party service could ask a BigchainDB network to determine if a particular user has permission to read some particular data. Indeed they could use BigchainDB to keep track of *all* the rights a user has for some data (not just the right to read it). That third party could also use BigchainDB to store audit logs, i.e. records of every read, write or other operation on stored data.

BigchainDB can be used in other ways to help parties exchange private data:

- It can be used to publicly disclose the *availability* of some private data (stored elsewhere). For example, there might be a description of the data and a price.

- It can be used to record the TLS handshakes which two parties sent to each other to establish an encrypted and authenticated TLS connection, which they could use to exchange private data with each other. (The stored handshake information wouldn't be enough, by itself, to decrypt the data.) It would be a "proof of TLS handshake."

- See the BigchainDB Privacy Protocols repository for more techniques.

1.11.4 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.