BigchainDB Documentation

Release

BigchainDB Contributors

Jan 16, 2018

Contents

BigchainDB is a scalable blockchain database. That is, it's a "big data" database with some blockchain characteristics added, including decentralization, immutability and native support for assets. You can read about the motivations, goals and high-level architecture in the BigchainDB whitepaper.

At a high level, one can communicate with a BigchainDB cluster (set of nodes) using the BigchainDB Client-Server 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?

BigchainDB is not production-ready. You can use it to build a prototype or proof-of-concept (POC); many people are already doing that. Once BigchainDB is production-ready, we'll make an announcement.

BigchainDB version numbers follow the conventions of *Semantic Versioning* as documented at semver.org. (For Python stuff, we use Python's version of Semantic Versioning.) This means, among other things:

- Before version 1.0, breaking API changes could happen in any new version, even in a change from version 0.Y.4 to 0.Y.5.
- Starting with version 1.0.0, breaking API changes will only happen when the MAJOR version changes (e.g. from 1.7.4 to 2.0.0, or from 4.9.3 to 5.0.0).

To review the release history of some particular BigchainDB software, go to the GitHub repository of that software and click on "Releases". For example, the release history of BigchainDB Server can be found at https://github.com/bigchaindb/bigchaindb/releases.

The BigchainDB Roadmap will give you a sense of the things we intend to do with BigchainDB in the near term and the long term.

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 set of closely-linked machines running MongoDB Server (or RethinkDB Server), 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 contains one logical MongoDB/RethinkDB datastore. 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 (e.g. purely vote-based, where each node gets a vote, and there are no special leadership roles).

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.

MongoDB and RethinkDB have an "admin" user which can't be deleted and which can make big changes to the database, such as dropping a table. Right now, that's a big security vulnerability, but we have plans to mitigate it by:

1. Locking down the admin user as much as possible.
2. Having all nodes inspect admin-type requests before acting on them. Requests can be checked against an evolving whitelist of allowed actions. Nodes requesting non-allowed requests can be removed from the list of cluster nodes.

It's worth noting that the admin user can't transfer assets, even today. The only way to create a valid transfer transaction is to fulfill the current (crypto) conditions on the asset, and the admin user can't do that because the admin user doesn't have the necessary private keys (or preimages, in the case of hashlock conditions). They're not stored in the database.

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. ("Enough" is usually a quorum.) 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. **Operating system diversity.** The servers should use a variety of operating systems, so that a security bug in one OS can't be used to exploit enough of the nodes.
5. **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.

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, blocks or votes) 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), each block is signed by the node that created it, and each vote is signed by the node that cast it.
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

While BigchainDB is not currently Byzantine fault tolerant (BFT), we plan to offer it as an option. Update Nov 2017: we're actively working on this, the next release or two will likely have support. More details to come in blog form and github issues

Related issue: Issue #293. We anticipate that turning on BFT will cause a dropoff in performance (for a gain in security).

In the meantime, there are practical things that one can do to increase security (e.g. firewalls, key management, and access controls).

1.7 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 strongly tamper-resistant; see *the page about immutability*.

1.7.1 BigchainDB Integration with Other Blockchains

BigchainDB works with the Interledger protocol, enabling the transfer of assets between BigchainDB and other blockchains, ledgers, and payment systems.

We're actively exploring ways that BigchainDB can be used with other blockchains and platforms.

Note: We used the word "owners" somewhat loosely above. A more accurate word might be fulfillers, signers, controllers, or transfer-enablers. See BigchainDB Server issue #626.

1.8 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 BigchainDB Server issue #626.

1.9 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.

1.9.1 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 documentation about Inputs and 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 BigchainDB Server issue #626.

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.)

1.9.2 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).

1.9.3 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.)

1.9.4 Example Transactions

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

1.10 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.

1.10.1 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 documentation about conditions in BigchainDB.

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.

1.10.2 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.

1.10.3 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.

1.10.4 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 (October 2017), doing so required the use of a plugin, so it's not possible using standard BigchainDB (which is what's available on IPDB). That may change in the future. If you're interested, contact BigchainDB.

1.11 Timestamps in BigchainDB

Each block and vote has an associated timestamp. Interpreting those timestamps is tricky, hence the need for this section.

1.11.1 Timestamp Sources & Accuracy

Timestamps in BigchainDB are provided by the node which created the block and the node that created the vote.

When a BigchainDB node needs a timestamp, it calls a BigchainDB utility function named `timestamp()`. There's a detailed explanation of how that function works below, but the short version is that it gets the Unix time from its system clock, rounded to the nearest second.

We advise BigchainDB nodes to run special software (an "NTP daemon") to keep their system clock in sync with standard time servers. (NTP stands for Network Time Protocol.)

1.11.2 Converting Timestamps to UTC

To convert a BigchainDB timestamp (a Unix time) to UTC, you need to know how the node providing the timestamp was set up. That's because different setups will report a different "Unix time" value around leap seconds! There's a

nice Red Hat Developer Blog post about the various setup options. If you want more details, see David Mills' pages about leap seconds, NTP, etc. (David Mills designed NTP.)

We advise BigchainDB nodes to run an NTP daemon with particular settings so that their timestamps are consistent.

If a timestamp comes from a node that's set up as we advise, it can be converted to UTC as follows:

1. Use a standard "Unix time to UTC" converter to get a UTC timestamp.
2. Is the UTC timestamp a leap second, or the second before/after a leap second? There's a list of all the leap seconds on Wikipedia.
3. If no, then you are done.
4. If yes, then it might not be possible to convert it to a single UTC timestamp. Even if it can't be converted to a single UTC timestamp, it *can* be converted to a list of two possible UTC timestamps. Showing how to do that is beyond the scope of this documentation. In all likelihood, you will never have to worry about leap seconds because they are very rare. (There were only 26 between 1972 and the end of 2015.)

1.11.3 Calculating Elapsed Time Between Two Timestamps

There's another gotcha with (Unix time) timestamps: you can't calculate the real-world elapsed time between two timestamps (correctly) by subtracting the smaller timestamp from the larger one. The result won't include any of the leap seconds that occured between the two timestamps. You could look up how many leap seconds happened between the two timestamps and add that to the result. There are many library functions for working with timestamps; those are beyond the scope of this documentation.

1.11.4 Interpreting Sets of Timestamps

You can look at many timestamps to get a statistical sense of when something happened. For example, a transaction in a decided-valid block has many associated timestamps:

- the timestamp of the block
- the timestamps of all the votes on the block

1.11.5 How BigchainDB Uses Timestamps

BigchainDB *doesn't* use timestamps to determine the order of transactions or blocks. In particular, the order of blocks is determined by MongoDB's oplog (or RethinkDB's changefeed) on the bigchain table.

BigchainDB does use timestamps for some things. When a Transaction is written to the backlog, a timestamp is assigned called the `assignment_timestamp`, to determine if it has been waiting in the backlog for too long (i.e. because the node assigned to it hasn't handled it yet).

1.11.6 Including Trusted Timestamps

If you want to create a transaction payload with a trusted timestamp, you can.

One way to do that would be to send a payload to a trusted timestamping service. They will send back a timestamp, a signature, and their public key. They should also explain how you can verify the signature. You can then include the original payload, the timestamp, the signature, and the service's public key in your transaction metadata. That way, anyone with the verification instructions can verify that the original payload was signed by the trusted timestamping service.

1.11.7 How the timestamp() Function Works

BigchainDB has a utility function named `timestamp()` which amounts to:

```
timestamp() = str(round(time.time()))
```

In other words, it calls the `time()` function in Python's `time` module, rounds that to the nearest integer, and converts the result to a string.

It rounds the output of `time.time()` to the nearest second because, according to the Python documentation for `time.time()`, "...not all systems provide time with a better precision than 1 second."

How does `time.time()` work? If you look in the C source code, it calls `floattime()` and `floattime()` calls clock_gettime(), if it's available.

```
ret = clock_gettime(CLOCK_REALTIME, &tp);
```

With `CLOCK_REALTIME` as the first argument, it returns the "Unix time." ("Unix time" is in quotes because its value around leap seconds depends on how the system is set up; see above.)

1.11.8 Why Not Use UTC, TAI or Some Other Time that Has Unambiguous Timestamps for Leap Seconds?

It would be nice to use UTC or TAI timestamps, but unfortunately there's no commonly-available, standard way to get always-accurate UTC or TAI timestamps from the operating system on typical computers today (i.e. accurate around leap seconds).

There *are* commonly-available, standard ways to get the "Unix time," such as clock_gettime() function available in C. That's what we use (indirectly via Python). ("Unix time" is in quotes because its value around leap seconds depends on how the system is set up; see above.)

The Unix-time-based timestamps we use are only ambiguous circa leap seconds, and those are very rare. Even for those timestamps, the extra uncertainty is only one second, and that's not bad considering that we only report timestamps to a precision of one second in the first place. All other timestamps can be converted to UTC with no ambiguity.