BigchainDB Server Documentation

Release 0.8.2

BigchainDB Contributors

Jan 27, 2017

Contents

1 Introduction 1

2 Quickstart 3

3 Cloud Deployment Starter Templates 5

4 Production Node Assumptions, Components & Requirements 11

5 Develop & Test BigchainDB Server 19

6 Settings & CLI 23

7 Drivers & Clients 29

8 Clusters & Federations 37

9 Topic Guides 49

10 Data Models 51

11 Transaction Schema 59

12 Release Notes 65

13 Appendices 67

Python Module Index 87

HTTP Routing Table 89

CHAPTER 1

Introduction

This is the documentation for BigchainDB Server, the BigchainDB software that one runs on servers (but not on clients).

If you want to use BigchainDB Server, then you should first understand what BigchainDB is, plus some of the specialized BigchaindB terminology. You can read about that in the overall BigchainDB project documentation.

Note that there are a few kinds of nodes:

- A **dev/test node** is a node created by a developer working on BigchainDB Server, e.g. for testing new or changed code. A dev/test node is typically run on the developer's local machine.
- A **bare-bones node** is a node deployed in the cloud, either as part of a testing cluster or as a starting point before upgrading the node to be production-ready. Our cloud deployment starter templates deploy a bare-bones node, as do our scripts for deploying a testing cluster on AWS.
- A **production node** is a node that is part of a federation's BigchainDB cluster. A production node has the most components and requirements.

1.1 Setup Instructions for Various Cases

- Set up a local stand-alone BigchainDB node for learning and experimenting: Quickstart
- Set up and run a bare-bones node in the cloud
- Set up and run a local dev/test node for developing and testing BigchainDB Server
- Deploy a testing cluster on AWS
- Set up and run a federation (including production nodes)

Instructions for setting up a client will be provided once there's a public test net.

1.2 Can I Help?

Yes! BigchainDB is an open-source project; we welcome contributions of all kinds. If you want to request a feature, file a bug report, make a pull request, or help in some other way, please see the CONTRIBUTING.md file.

CHAPTER 2

Quickstart

This page has instructions to set up a single stand-alone BigchainDB node for learning or experimenting. Instructions for other cases are elsewhere. We will assume you're using Ubuntu 14.04 or similar. If you're not using Linux, then you might try running BigchainDB with Docker.

A. Install RethinkDB Server

B. Open a Terminal and run RethinkDB Server with the command:

```
rethinkdb
```

C. Ubuntu 14.04 already has Python 3.4, so you don't need to install it, but you do need to install a couple other things:

```
sudo apt-get update
sudo apt-get install g++ python3-dev libffi-dev
```

D. Get the latest version of pip and setuptools:

```
sudo apt-get install python3-pip
sudo pip3 install --upgrade pip setuptools
```

E. Install the `bigchaindb` Python package from PyPI:

```
sudo pip3 install bigchaindb
```

F. Configure and run BigchainDB Server:

```
bigchaindb -y configure
bigchaindb start
```

That's it!

Next Steps: You could build valid transactions and push them to your running BigchainDB Server using the BigchaindB Python Driver.

CHAPTER 3

Cloud Deployment Starter Templates

We have some "starter templates" to deploy a basic, working, but bare-bones BigchainDB node on various cloud providers. They should *not* be used as-is to deploy a node for production. They can be used as a starting point. A full production node should meet the requirements outlined in the section on *production node assumptions, components and requirements*.

You don't have to use the tools we use in the starter templates (e.g. Terraform and Ansible). You can use whatever tools you prefer.

If you find the cloud deployment starter templates for nodes helpful, then you may also be interested in our scripts for *deploying a testing cluster on AWS* (documented in the Clusters & Federations section).

3.1 Template: Using Terraform to Provision an Ubuntu Machine on AWS

If you didn't read the introduction to the cloud deployment starter templates, please do that now. The main point is that they're not for deploying a production node; they can be used as a starting point.

This page explains a way to use Terraform to provision an Ubuntu machine (i.e. an EC2 instance with Ubuntu 14.04) and other resources on AWS. That machine can then be used to host a one-machine BigchainDB node.

3.1.1 Install Terraform

The Terraform documentation has installation instructions for all common operating systems.

If you don't want to run Terraform on your local machine, you can install it on a cloud machine under your control (e.g. on AWS).

Note: Hashicorp has an enterprise version of Terraform called "Terraform Enterprise." You can license it by itself or get it as part of Atlas. If you decide to license Terraform Enterprise or Atlas, be sure to install it on your own hosting (i.e. "on premise"), not on the hosting provided by Hashicorp. The reason is that BigchainDB clusters are supposed to be decentralized. If everyone used Hashicorp's hosted Atlas, then that would be a point of centralization.

Ubuntu Installation Tips

If you want to install Terraform on Ubuntu, first download the .zip file. Then install it in `/opt`:

```
sudo mkdir -p /opt/terraform
sudo unzip path/to/zip-file.zip -d /opt/terraform
```

Why install it in `/opt`? See the answers at Ask Ubuntu.

Next, add `/opt/terraform` to your path. If you use bash for your shell, then you could add this line to `~/.bashrc`:

```
export PATH="/opt/terraform:$PATH"
```

After doing that, relaunch your shell or force it to read `~/.bashrc` again, e.g. by doing `source ~/.bashrc`. You can verify that terraform is installed and in your path by doing:

```
terraform --version
```

It should say the current version of Terraform.

3.1.2 Get Set Up to Use Terraform

First, do the basic AWS setup steps outlined in the Appendices.

Then go to the `.../bigchaindb/ntools/one-m/aws/` directory and open the file `variables.tf`. Most of the variables have sensible default values, but you can change them if you like. In particular, you may want to change `aws_region`. (Terraform looks in `~/.aws/credentials` to get your AWS credentials, so you don't have to enter those anywhere.)

The `ssh_key_name` has no default value, so Terraform will prompt you every time it needs it.

To see what Terraform will do, run:

```
terraform plan
```

It should ask you the value of `ssh_key_name`.

It figured out the plan by reading all the `.tf` Terraform files in the directory.

If you don't want to be asked for the `ssh_key_name`, you can change the default value of `ssh_key_name` (in the file `variables.tf`) or you can set an environmen variable named `TF_VAR_ssh_key_name`.

3.1.3 Use Terraform to Provision Resources

To provision all the resources specified in the plan, do the following. **Note: This will provision actual resources on AWS, and those cost money. Be sure to shut down the resources you don't want to keep running later, otherwise the cost will keep growing.**

```
terraform apply
```

Terraform will report its progress as it provisions all the resources. Once it's done, you can go to the Amazon EC2 web console and see the instance, its security group, its elastic IP, and its attached storage volumes (one for the root directory and one for RethinkDB storage).

At this point, there is no software installed on the instance except for Ubuntu 14.04 and whatever else came with the Amazon Machine Image (AMI) specified in the Terraform configuration (files).

The next step is to install, configure and run all the necessary software for a BigchainDB node. You could use our example Ansible playbook to do that.

3.1.4 Optional: "Destroy" the Resources

If you want to shut down all the resources just provisioned, you must first disable termination protection on the instance:

1. Go to the EC2 console and select the instance you just launched. It should be named `BigchainDB_node`.
2. Click **Actions** > **Instance Settings** > **Change Termination Protection** > **Yes, Disable**
3. Back in your terminal, do `terraform destroy`

Terraform should "destroy" (i.e. terminate or delete) all the AWS resources you provisioned above.

If it fails (e.g. because of an attached and mounted EBS volume), then you can terminate the instance using the EC2 console: **Actions** > **Instance State** > **Terminate** > **Yes, Terminate**. Once the instance is terminated, you should still do `terraform destroy` to make sure that all the other resources are destroyed.

3.2 Template: Ansible Playbook to Run a BigchainDB Node on an Ubuntu Machine

If you didn't read the introduction to the cloud deployment starter templates, please do that now. The main point is that they're not for deploying a production node; they can be used as a starting point.

This page explains how to use Ansible to install, configure and run all the software needed to run a one-machine BigchainDB node on a server running Ubuntu 14.04.

3.2.1 Install Ansible

The Ansible documentation has installation instructions. Note the control machine requirements: at the time of writing, Ansible required Python 2.6 or 2.7. (Support for Python 3 is a goal of Ansible 2.2.)

For example, you could create a special Python 2.x virtualenv named `ansenv` and then install Ansible in it:

```
cd repos/bigchaindb/ntools
virtualenv -p /usr/local/lib/python2.7.11/bin/python ansenv
source ansenv/bin/activate
pip install ansible
```

3.2.2 About Our Example Ansible Playbook

Our example Ansible playbook installs, configures and runs a basic BigchainDB node on an Ubuntu 14.04 machine. That playbook is in `.../bigchaindb/ntools/one-m/ansible/one-m-node.yml`.

When you run the playbook (as per the instructions below), it ensures all the necessary software is installed, configured and running. It can be used to get a BigchainDB node set up on a bare Ubuntu 14.04 machine, but it can also be used to ensure that everything is okay on a running BigchainDB node. (If you run the playbook against a host where everything is okay, then it won't change anything on that host.)

3.2.3 Create an Ansible Inventory File

An Ansible "inventory" file is a file which lists all the hosts (machines) you want to manage using Ansible. (Ansible will communicate with them via SSH.) Right now, we only want to manage one host.

First, determine the public IP address of the host (i.e. something like `192.0.2.128`).

Then create a one-line text file named `hosts` by doing this:

```
# cd to the directory .../bigchaindb/ntools/one-m/ansible
echo "192.0.2.128" > hosts
```

but replace `192.0.2.128` with the IP address of the host.

3.2.4 Run the Ansible Playbook

The next step is to run the Ansible playbook named `one-m-node.yml`:

```
# cd to the directory .../bigchaindb/ntools/one-m/ansible
ansible-playbook -i hosts --private-key ~/.ssh/<key-name> one-m-node.yml
```

where `<key-name>` should be replaced by the name of the SSH private key you created earlier (for SSHing to the host machine at your cloud hosting provider).

What did you just do? Running that playbook ensures all the software necessary for a one-machine BigchainDB node is installed, configured, and running properly. You can run that playbook on a regular schedule to ensure that the system stays properly configured. If something is okay, it does nothing; it only takes action when something is not as-desired.

3.2.5 Some Notes on the One-Machine Node You Just Got Running

- It ensures that the installed version of RethinkDB is `2.3.4~0trusty`. You can change that by changing the installation task.
- It uses a very basic RethinkDB configuration file based on `bigchaindb/ntools/one-m/ansible/roles/rethinkdb/templates/rethinkdb.conf.j2`.
- If you edit the RethinkDB configuration file, then running the Ansible playbook will **not** restart RethinkDB for you. You must do that manually. (You can stop RethinkDB using `sudo /etc/init.d/rethinkdb stop`; run the playbook to get RethinkDB started again. This assumes you're using init.d, which is what the Ansible playbook assumes. If you want to use systemd, you'll have to edit the playbook accordingly, and stop RethinkDB using `sudo systemctl stop rethinkdb@<name_instance>`.)
- It generates and uses a default BigchainDB configuration file, which it stores in `~/.bigchaindb` (the default location).
- If you edit the BigchainDB configuration file, then running the Ansible playbook will **not** restart BigchainDB for you. You must do that manually. (You could stop it using `sudo killall -9 bigchaindb`. Run the playbook to get it started again.)

3.2.6 Optional: Create an Ansible Config File

The above command (`ansible-playbook -i ...`) is fairly long. You can omit the optional arguments if you put their values in an Ansible configuration file (config file) instead. There are many places where you can put a config file, but to make one specifically for the "one-m" case, you should put it in `.../bigchaindb/ntools/one-m/ansible/`. In that directory, create a file named `ansible.cfg` with the following contents:

```
[defaults]
private_key_file = $HOME/.ssh/<key-name>
inventory = hosts
```

where, as before, `<key-name>` must be replaced.

3.2.7 Next Steps

You could make changes to the Ansible playbook (and the resources it uses) to make the node more production-worthy. See the section on production node assumptions, components and requirements.

3.3 Template: Node Deployment on Azure

If you didn't read the introduction to the cloud deployment starter templates, please do that now. The main point is that they're not for deploying a production node; they can be used as a starting point.

One can deploy a BigchainDB node on Azure using the template in Microsoft's azure-quickstart-templates repository on GitHub:

1. Go to the /blockchain subdirectory in that repository.
2. Click the button labelled **Deploy to Azure**.
3. If you're not already logged in to Microsoft Azure, then you'll be prompted to login. If you don't have an account, then you'll have to create one.
4. One you are logged in to the Microsoft Azure Portal, you should be taken to a form titled **Blockchain Template**. Below there are some notes to help with filling in that form.
5. Deployment takes a few minutes. You can follow the notifications by clicking the bell icon at the top of the screen. When done, you should see a notification saying "Deployment to resource group '[your resource group name]' was successful." The install script (`bigchaindb.sh`) installed RethinkDB, configured it using the default RethinkDB config file, and ran it. It also used pip to install the latest `bigchaindb` from PyPI.
6. Find out the public IP address of the virtual machine in the Azure Portal. Example: `40.69.87.250`
7. ssh in to the virtual machine at that IP address, e.g. `ssh adminusername@40.69.87.250` (where `adminusername` should be replaced by the Admin Username you entered into the form, and `40.69.87.250` should be replaced by the IP address you found in the last step).
8. You should be prompted for a password. Enter the Admin Password you entered into the form.
9. Configure BigchainDB using the default BigchainDB settings: `bigchaindb -y configure`
10. Run BigchainDB: `bigchaindb start`

BigchainDB should now be running on the Azure VM.

Remember to shut everything down when you're done (via the Azure Portal), because it generally costs money to run stuff on Azure.

3.3.1 Notes on the Blockchain Template Form Fields

Resource group - You can use an existing resource group (if you have one) or create a new one named whatever you like, but avoid using fancy characters in the name because Azure might have problems if you do.

Location is the Microsoft Azure data center where you want the BigchainDB node to run. Pick one close to where you are located.

Vm Dns Prefix - Once your virtual machine (VM) is deployed, it will have a public IP address and a DNS name (hostname) something like `DNSprefix.northeurope.cloudapp.azure.com`. The `DNSprefix` will be whatever you enter into this field.

You can use whatever **Admin Username** and **Admin Password** you like (provided you don't get too fancy). It will complain if your password is too simple. You'll need these later to ssh into the VM.

Blockchain Software - Select `bigchaindb`.

For **Vm Size**, select `Standard_D1_v2` or better.

_artifacts Location - Leave this alone.

_artifacts Location Sas Token - Leave this alone (blank).

Don't forget to scroll down and check the box to agree to the terms and conditions.

Once you've finished the form, click the button labelled **Purchase**. (Generally speaking, it costs money to run stuff on Azure.)

CHAPTER 4

Production Node Assumptions, Components & Requirements

4.1 Production Node Assumptions

If you're not sure what we mean by a BigchainDB *node*, *cluster*, *federation*, or *production node*, then see the section in the Introduction where we defined those terms.

We make some assumptions about production nodes:

1. **Each production node is set up and managed by an experienced professional system administrator (or a team of them).**
2. Each production node in a federation's cluster is managed by a different person or team.

Because of the first assumption, we don't provide a detailed cookbook explaining how to secure a server, or other things that a sysadmin should know. (We do provide some starter templates, but those are just a starting point.)

4.2 Production Node Components

A BigchainDB node must include, at least:

- BigchainDB Server and
- RethinkDB Server.

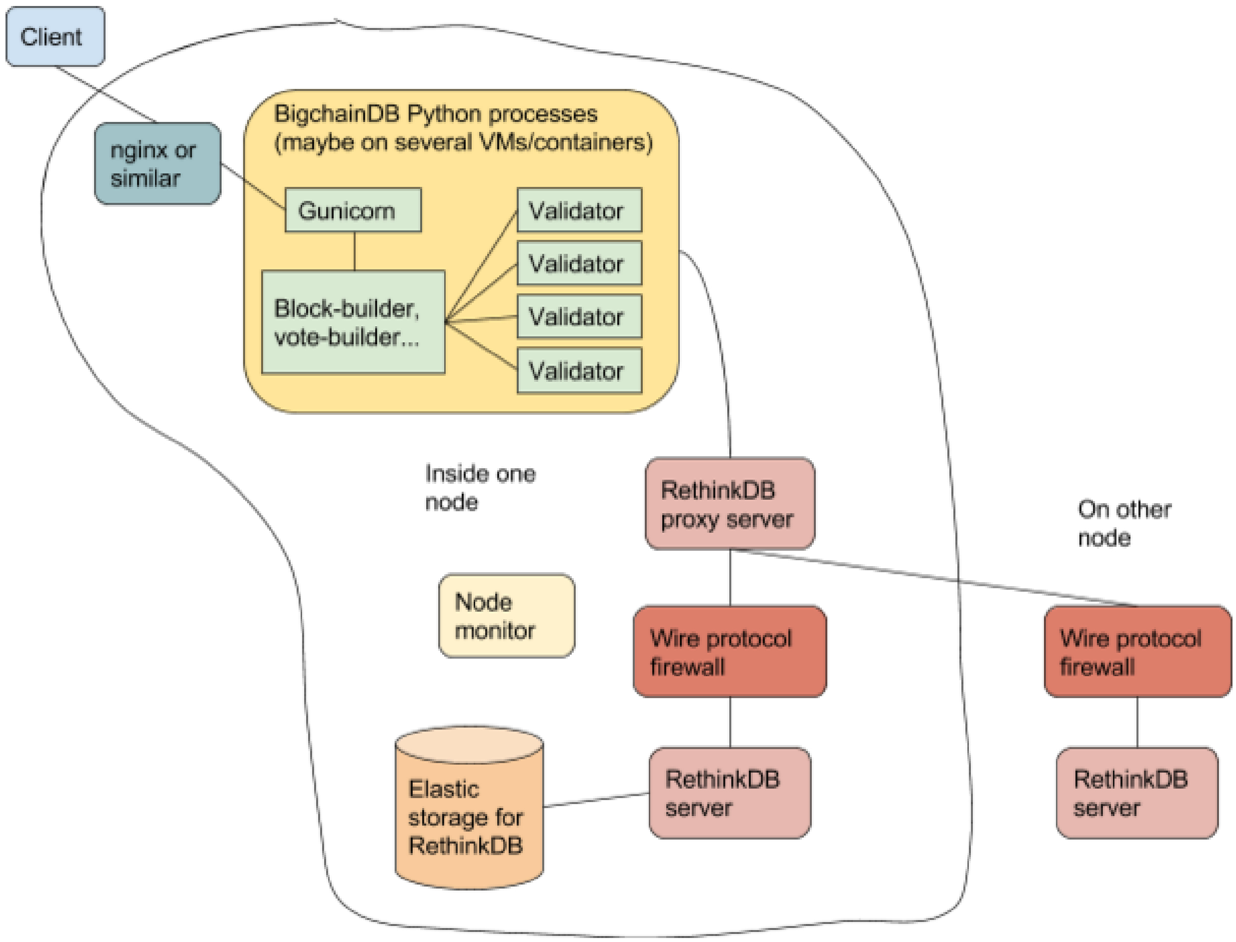
When doing development and testing, it's common to install both on the same machine, but in a production environment, it may make more sense to install them on separate machines.

In a production environment, a BigchainDB node should have several other components, including:

- nginx or similar, as a reverse proxy and/or load balancer for the Gunicorn server(s) inside the node
- An NTP daemon running on all machines running BigchainDB code, and possibly other machines
- A RethinkDB proxy server
- A RethinkDB "wire protocol firewall" (in the future: this component doesn't exist yet)

- Scalable storage for RethinkDB (e.g. using RAID)
- Monitoring software, to monitor all the machines in the node
- Configuration management agents (if you're using a configuration managment system that uses agents)
- Maybe more

The relationship between these components is illustrated below.



4.3 Production Node Requirements

Note: This section will be broken apart into several pages, e.g. NTP requirements, RethinkDB requirements, BigchainDB requirements, etc. and those pages will add more details.

4.3.1 OS Requirements

- RethinkDB Server will run on any modern OS. Note that the Fedora package isn't officially supported. Also, official support for Windows is fairly recent (April 2016).
- BigchainDB Server requires Python 3.4+ and Python 3.4+ will run on any modern OS.

- BigchaindB Server uses the Python `multiprocessing` package and some functionality in the `multiprocessing` package doesn't work on OS X. You can still use Mac OS X if you use Docker or a virtual machine.

The BigchainDB core dev team uses Ubuntu 14.04, Ubuntu 16.04, Fedora 23, and Fedora 24.

We don't test BigchainDB on Windows or Mac OS X, but you can try.

- If you run into problems on Windows, then you may want to try using Vagrant. One of our community members (@Mec-Is) wrote a page about how to install BigchainDB on a VM with Vagrant.
- If you have Mac OS X and want to experiment with BigchainDB, then you could do that using Docker.

4.3.2 Storage Requirements

When it comes to storage for RethinkDB, there are many things that are nice to have (e.g. SSDs, high-speed input/output [IOPS], replication, reliability, scalability, pay-for-what-you-use), but there are few *requirements* other than:

1. have enough storage to store all your data (and its replicas), and
2. make sure your storage solution (hardware and interconnects) can handle your expected read & write rates.

For RethinkDB's failover mechanisms to work, every RethinkDB table must have at least three replicas (i.e. a primary replica and two others). For example, if you want to store 10 GB of unique data, then you need at least 30 GB of storage. (Indexes and internal metadata are stored in RAM.)

As for the read & write rates, what do you expect those to be for your situation? It's not enough for the storage system alone to handle those rates: the interconnects between the nodes must also be able to handle them.

4.3.3 Memory (RAM) Requirements

In their FAQ, RethinkDB recommends that, "RethinkDB servers have at least 2GB of RAM..." (source)

In particular: "RethinkDB requires data structures in RAM on each server proportional to the size of the data on that server's disk, usually around 1% of the size of the total data set." (source) We asked what they meant by "total data set" and they said it's "referring to only the data stored on the particular server."

Also, "The storage engine is used in conjunction with a custom, B-Tree-aware caching engine which allows file sizes many orders of magnitude greater than the amount of available memory. RethinkDB can operate on a terabyte of data with about ten gigabytes of free RAM." (source) (In this case, it's the *cluster* which has a total of one terabyte of data, and it's the *cluster* which has a total of ten gigabytes of RAM. That is, if you add up the RethinkDB RAM on all the servers, it's ten gigabytes.)

In reponse to our questions about RAM requirements, @danielmewes (of RethinkDB) wrote:

> ... If you replicate the data, the amount of data per server increases accordingly, because multiple copies of the same data will be held by different servers in the cluster.

For example, if you increase the data replication factor from 1 to 2 (i.e. the primary plus one copy), then that will double the RAM needed for metadata. Also from @danielmewes:

> **For reasonable performance, you should probably aim at something closer to 5-10% of the data size.** [Emphasis added] The 1% is the bare minimum and doesn't include any caching. If you want to run near the minimum, you'll also need to manually lower RethinkDB's cache size through the `--cache-size` parameter to free up enough RAM for the metadata overhead...

RethinkDB has documentation about its memory requirements. You can use that page to get a better estimate of how much memory you'll need. In particular, note that RethinkDB automatically configures the cache size limit to be about half the available memory, but it can be no lower than 100 MB. As @danielmewes noted, you can manually change the cache size limit (e.g. to free up RAM for queries, metadata, or other things).

If a RethinkDB process (on a server) runs out of RAM, the operating system will start swapping RAM out to disk, slowing everything down. According to @danielmewes:

> Going into swap is usually pretty bad for RethinkDB, and RethinkDB servers that have gone into swap often become so slow that other nodes in the cluster consider them unavailable and terminate the connection to them. I recommend adjusting RethinkDB's cache size conservatively to avoid this scenario. RethinkDB will still make use of additional RAM through the operating system's block cache (though less efficiently than when it can keep data in its own cache).

4.3.4 Filesystem Requirements

RethinkDB "supports most commonly used file systems" (source) but it has issues with BTRFS (B-tree file system).

It's best to use a filesystem that supports direct I/O, because that will improve RethinkDB performance (if you tell RethinkDB to use direct I/O). Many compressed or encrypted filesystems don't support direct I/O.

4.4 Set Up and Run a Cluster Node

This is a page of general guidelines for setting up a production node. It says nothing about how to upgrade software, storage, processing, etc. or other details of node management. It will be expanded more in the future.

4.4.1 Get a Server

The first step is to get a server (or equivalent) which meets the requirements for a BigchainDB node.

4.4.2 Secure Your Server

The steps that you must take to secure your server depend on your server OS and where your server is physically located. There are many articles and books about how to secure a server. Here we just cover special considerations when securing a BigchainDB node.

There are some notes on BigchainDB-specific firewall setup in the Appendices.

4.4.3 Sync Your System Clock

A BigchainDB node uses its system clock to generate timestamps for blocks and votes, so that clock should be kept in sync with some standard clock(s). The standard way to do that is to run an NTP daemon (Network Time Protocol daemon) on the node. (You could also use tlsdate, which uses TLS timestamps rather than NTP, but don't: it's not very accurate and it will break with TLS 1.3, which removes the timestamp.)

NTP is a standard protocol. There are many NTP daemons implementing it. We don't recommend a particular one. On the contrary, we recommend that different nodes in a federation run different NTP daemons, so that a problem with one daemon won't affect all nodes.

Please see the notes on NTP daemon setup in the Appendices.

4.4.4 Set Up Storage for RethinkDB Data

Below are some things to consider when setting up storage for the RethinkDB data. The Appendices have a section with concrete examples.

We suggest you set up a separate storage "device" (partition, RAID array, or logical volume) to store the RethinkDB data. Here are some questions to ask:

- How easy will it be to add storage in the future? Will I have to shut down my server?
- How big can the storage get? (Remember that RAID can be used to make several physical drives look like one.)
- How fast can it read & write data? How many input/output operations per second (IOPS)?
- How does IOPS scale as more physical hard drives are added?
- What's the latency?
- What's the reliability? Is there replication?
- What's in the Service Level Agreement (SLA), if applicable?
- What's the cost?

There are many options and tradeoffs. Don't forget to look into Amazon Elastic Block Store (EBS) and Amazon Elastic File System (EFS), or their equivalents from other providers.

Storage Notes Specific to RethinkDB

- The RethinkDB storage engine has a number of SSD optimizations, so you *can* benefit from using SSDs. (source)
- If you want a RethinkDB cluster to store an amount of data D, with a replication factor of R (on every table), and the cluster has N nodes, then each node will need to be able to store R×D/N data.
- RethinkDB tables can have at most 64 shards. For example, if you have only one table and more than 64 nodes, some nodes won't have the primary of any shard, i.e. they will have replicas only. In other words, once you pass 64 nodes, adding more nodes won't provide more storage space for new data. If the biggest single-node storage available is d, then the most you can store in a RethinkDB cluster is < 64×d: accomplished by putting one primary shard in each of 64 nodes, with all replica shards on other nodes. (This is assuming one table. If there are T tables, then the most you can store is < 64×d×T.)
- When you set up storage for your RethinkDB data, you may have to select a filesystem. (Sometimes, the filesystem is already decided by the choice of storage.) We recommend using a filesystem that supports direct I/O (Input/Output). Many compressed or encrypted file systems don't support direct I/O. The ext4 filesystem supports direct I/O (but be careful: if you enable the data=journal mode, then direct I/O support will be disabled; the default is data=ordered). If your chosen filesystem supports direct I/O and you're using Linux, then you don't need to do anything to request or enable direct I/O. RethinkDB does that.
- RethinkDB stores its data in a specific directory. You can tell RethinkDB *which* directory using the RethinkDB config file, as explained below. In this documentation, we assume the directory is `/data`. If you set up a separate device (partition, RAID array, or logical volume) to store the RethinkDB data, then mount that device on `/data`.

4.4.5 Install RethinkDB Server

If you don't already have RethinkDB Server installed, you must install it. The RethinkDB documentation has instructions for how to install RethinkDB Server on a variety of operating systems.

4.4.6 Configure RethinkDB Server

Create a RethinkDB configuration file (text file) named `instance1.conf` with the following contents (explained below):

```
directory=/data
bind=all
direct-io
# Replace node?_hostname with actual node hostnames below, e.g. rdb.examples.com
join=node0_hostname:29015
join=node1_hostname:29015
join=node2_hostname:29015
# continue until there's a join= line for each node in the federation
```

- `directory=/data` tells the RethinkDB node to store its share of the database data in `/data`.
- `bind=all` binds RethinkDB to all local network interfaces (e.g. loopback, Ethernet, wireless, whatever is available), so it can communicate with the outside world. (The default is to bind only to local interfaces.)
- `direct-io` tells RethinkDB to use direct I/O (explained earlier). Only include this line if your file system supports direct I/O.
- `join=hostname:29015` lines: A cluster node needs to find out the hostnames of all the other nodes somehow. You *could* designate one node to be the one that every other node asks, and put that node's hostname in the config file, but that wouldn't be very decentralized. Instead, we include *every* node in the list of nodes-to-ask.

If you're curious about the RethinkDB config file, there's a RethinkDB documentation page about it. The explanations of the RethinkDB command-line options are another useful reference.

See the RethinkDB documentation on securing your cluster.

4.4.7 Install Python 3.4+

If you don't already have it, then you should install Python 3.4+.

If you're testing or developing BigchainDB on a stand-alone node, then you should probably create a Python 3.4+ virtual environment and activate it (e.g. using virtualenv or conda). Later we will install several Python packages and you probably only want those installed in the virtual environment.

4.4.8 Install BigchainDB Server

BigchainDB Server has some OS-level dependencies that must be installed.

On Ubuntu 14.04, we found that the following was enough:

```
sudo apt-get update
sudo apt-get install g++ python3-dev libffi-dev
```

On Fedora 23, we found that the following was enough (tested in February 2015):

```
sudo dnf update
sudo dnf install gcc-c++ redhat-rpm-config python3-devel libffi-devel
```

(If you're using a version of Fedora before version 22, you may have to use `yum` instead of `dnf`.)

With OS-level dependencies installed, you can install BigchainDB Server with `pip` or from source.

How to Install BigchainDB with pip

BigchainDB (i.e. both the Server and the officially-supported drivers) is distributed as a Python package on PyPI so you can install it using `pip`. First, make sure you have an up-to-date Python 3.4+ version of `pip` installed:

```
pip -V
```

If it says that `pip` isn't installed, or it says `pip` is associated with a Python version less than 3.4, then you must install a `pip` version associated with Python 3.4+. In the following instructions, we call it `pip3` but you may be able to use `pip` if that refers to the same thing. See the `pip` installation instructions.

On Ubuntu 14.04, we found that this works:

```
sudo apt-get install python3-pip
```

That should install a Python 3 version of `pip` named `pip3`. If that didn't work, then another way to get `pip3` is to do `sudo apt-get install python3-setuptools` followed by `sudo easy_install3 pip`.

You can upgrade `pip` (`pip3`) and `setuptools` to the latest versions using:

```
pip3 install --upgrade pip setuptools
pip3 -V
```

Now you can install BigchainDB Server (and officially-supported BigchainDB drivers) using:

```
pip3 install bigchaindb
```

(If you're not in a virtualenv and you want to install bigchaindb system-wide, then put `sudo` in front.)

Note: You can use `pip3` to upgrade the `bigchaindb` package to the latest version using `pip3 install --upgrade bigchaindb`.

How to Install BigchainDB from Source

If you want to install BitchainDB from source because you want to use the very latest bleeding-edge code, clone the public repository:

```
git clone git@github.com:bigchaindb/bigchaindb.git
python setup.py install
```

4.4.9 Configure BigchainDB Server

Start by creating a default BigchainDB config file:

```
bigchaindb -y configure
```

(There's documentation for the `bigchaindb` command is in the section on the BigchainDB Command Line Interface (CLI).)

Edit the created config file:

- Open `$HOME/.bigchaindb` (the created config file) in your text editor.
- Change `"server": {"bind": "localhost:9984", ... }` to `"server": {"bind": "0.0.0.0:9984", ... }`. This makes it so traffic can come from any IP address to port 9984 (the HTTP Client-Server API port).

- Change `"api_endpoint": "http://localhost:9984/api/v1"` to `"api_endpoint": "http://your_api_hostname:9984/api/v1"`
- Change `"keyring": []` to `"keyring": ["public_key_of_other_node_A", "public_key_of_other_node_B", "..."]` i.e. a list of the public keys of all the other nodes in the federation. The keyring should *not* include your node's public key.

For more information about the BigchainDB config file, see Configuring a BigchainDB Node.

4.4.10 Run RethinkDB Server

Start RethinkDB using:

```
rethinkdb --config-file path/to/instance1.conf
```

except replace the path with the actual path to `instance1.conf`.

Note: It's possible to make RethinkDB start at system startup.

You can verify that RethinkDB is running by opening the RethinkDB web interface in your web browser. It should be at `http://rethinkdb-hostname:8080/`. If you're running RethinkDB on localhost, that would be http://localhost:8080/.

4.4.11 Run BigchainDB Server

After all node operators have started RethinkDB, but before they start BigchainDB, one designated node operator must configure the RethinkDB database by running the following commands:

```
bigchaindb init
bigchaindb set-shards numshards
bigchaindb set-replicas numreplicas
```

where:

- `bigchaindb init` creates the database within RethinkDB, the tables, the indexes, and the genesis block.
- `numshards` should be set to the number of nodes in the initial cluster.
- `numreplicas` should be set to the database replication factor decided by the federation. It must be 3 or more for RethinkDB failover to work.

Once the RethinkDB database is configured, every node operator can start BigchainDB using:

```
bigchaindb start
```

CHAPTER 5

Develop & Test BigchainDB Server

5.1 Set Up & Run a Dev/Test Node

This page explains how to set up a minimal local BigchainDB node for development and testing purposes.

The BigchainDB core dev team develops BigchainDB on recent Ubuntu and Fedora distributions, so we recommend you use one of those. BigchainDB Server doesn't work on Windows and Mac OS X (unless you use a VM or containers).

5.1.1 Option A: Using a Local Dev Machine

First, read through the BigchainDB CONTRIBUTING.md file. It outlines the steps to setup a machine for developing and testing BigchainDB.

Next, create a default BigchainDB config file (in `$HOME/.bigchaindb`):

```
bigchaindb -y configure
```

Note: The BigchainDB CLI and the BigchainDB Configuration Settings are documented elsewhere. (Click the links.)

Start RethinkDB using:

```
rethinkdb
```

You can verify that RethinkDB is running by opening the RethinkDB web interface in your web browser. It should be at http://localhost:8080/.

To run BigchainDB Server, do:

```
bigchaindb start
```

You can run all the unit tests to test your installation.

The BigchainDB CONTRIBUTING.md file has more details about how to contribute.

5.1.2 Option B: Using a Dev Machine on Cloud9

Ian Worrall of Encrypted Labs wrote a document (PDF) explaining how to set up a BigchainDB (Server) dev machine on Cloud9:

Download that document from GitHub

5.1.3 Option C: Using a Local Dev Machine and Docker

You need to have recent versions of docker engine and docker-compose.

Build the images:

```
docker-compose build
```

Start RethinkDB:

```
docker-compose up -d rdb
```

The RethinkDB web interface should be accessible at http://localhost:58080/. Depending on which platform, and/or how you are running docker, you may need to change `localhost` for the `ip` of the machine that is running docker. As a dummy example, if the `ip` of that machine was `0.0.0.0`, you would accees the web interface at: http://0.0.0.0:58080/.

Start a BigchainDB node:

```
docker-compose up -d bdb
```

You can monitor the logs:

```
docker-compose logs -f bdb
```

If you wish to run the tests:

```
docker-compose run --rm bdb py.test -v -n auto
```

A quick check to make sure that the BigchainDB server API is operational:

```
curl $(docker-compose port bdb 9984)
```

should give you something like:

```
{
  "api_endpoint": "http://bdb:9984/api/v1",
  "keyring": [],
  "public_key": "Brx8g4DdtEhccsENzNNV6yvQHR8s9ebhKyXPFkWUXh5e",
  "software": "BigchainDB",
  "version": "0.6.0"
}
```

How does the above curl command work? Inside the Docker container, BigchainDB exposes the HTTP API on port `9984`. First we get the public port where that port is bound:

```
docker-compose port bdb 9984
```

The port binding will change whenever you stop/restart the `bdb` service. You should get an output similar to:

```
0.0.0.0:32772
```

but with a port different from `32772`.

Knowing the public port we can now perform a simple `GET` operation against the root:

```
curl 0.0.0.0:32772
```

5.2 Running Unit Tests

Once you've installed BigchainDB Server, you may want to run all the unit tests. This section explains how.

First of all, if you installed BigchainDB Server using `pip` (i.e. by getting the package from PyPI), then you didn't install the tests. **Before you can run all the unit tests, you must install BigchainDB from source.**

To run all the unit tests, first make sure you have RethinkDB running:

```
$ rethinkdb
```

then in another terminal, do:

```
$ python setup.py test
```

(Aside: How does the above command work? The documentation for pytest-runner explains. We use pytest to write all unit tests.)

5.2.1 Using docker-compose to Run the Tests

You can also use `docker-compose` to run the unit tests.

Start `RethinkDB` in the background:

```
$ docker-compose up -d rdb
```

then run the unit tests using:

```
$ docker-compose run --rm bdb py.test -v
```

CHAPTER 6

Settings & CLI

6.1 Configuration Settings

The value of each BigchainDB Server configuration setting is determined according to the following rules:

- If it's set by an environment variable, then use that value
- Otherwise, if it's set in a local config file, then use that value
- Otherwise, use the default value

For convenience, here's a list of all the relevant environment variables (documented below):

`BIGCHAINDB_KEYPAIR_PUBLIC` `BIGCHAINDB_KEYPAIR_PRIVATE` `BIGCHAINDB_KEYRING` `BIGCHAINDB_DATABASE_HOST` `BIGCHAINDB_DATABASE_PORT` `BIGCHAINDB_DATABASE_NAME` `BIGCHAINDB_SERVER_BIND` `BIGCHAINDB_SERVER_WORKERS` `BIGCHAINDB_SERVER_THREADS` `BIGCHAINDB_API_ENDPOINT` `BIGCHAINDB_STATSD_HOST` `BIGCHAINDB_STATSD_PORT` `BIGCHAINDB_STATSD_RATE` `BIGCHAINDB_CONFIG_PATH` `BIGCHAINDB_BACKLOG_REASSIGN_DELAY`

The local config file is `$HOME/.bigchaindb` by default (a file which might not even exist), but you can tell BigchainDB to use a different file by using the `-c` command-line option, e.g. `bigchaindb -c path/to/config_file.json start` or using the `BIGCHAINDB_CONFIG_PATH` environment variable, e.g. `BIGHAINDB_CONFIG_PATH=.my_bigchaindb_config bigchaindb start`. Note that the `-c` command line option will always take precedence if both the `BIGCHAINDB_CONFIG_PATH` and the `-c` command line option are used.

You can read the current default values in the file bigchaindb/__init__.py. (The link is to the latest version.)

Running `bigchaindb -y configure` will generate a local config file in `$HOME/.bigchaindb` with all the default values, with two exceptions: It will generate a valid private/public keypair, rather than using the default keypair (`None` and `None`).

6.1.1 keypair.public & keypair.private

The cryptographic keypair used by the node. The public key is how the node idenifies itself to the world. The private key is used to generate cryptographic signatures. Anyone with the public key can verify that the signature was generated by whoever had the corresponding private key.

Example using environment variables

```
export BIGCHAINDB_KEYPAIR_PUBLIC=8wHUvvraRo5yEoJAt66UTZaFq9YZ9tFFwcauKPDtjkGw
export BIGCHAINDB_KEYPAIR_PRIVATE=5C5Cknco7YxBRP9AgB1cbUVTL4FAcooxErLygw1DeG2D
```

Example config file snippet

```
"keypair": {
  "public": "8wHUvvraRo5yEoJAt66UTZaFq9YZ9tFFwcauKPDtjkGw",
  "private": "5C5Cknco7YxBRP9AgB1cbUVTL4FAcooxErLygw1DeG2D"
}
```

Internally (i.e. in the Python code), both keys have a default value of `None`, but that's not a valid key. Therefore you can't rely on the defaults for the keypair. If you want to run BigchainDB, you must provide a valid keypair, either in the environment variables or in the local config file. You can generate a local config file with a valid keypair (and default everything else) using `bigchaindb -y configure`.

6.1.2 keyring

A list of the public keys of all the nodes in the cluster, excluding the public key of this node.

Example using an environment variable

```
export BIGCHAINDB_
↪KEYRING=BnCsre9MPBeQK8QZBFznU2dJJ2GwtvnSMdemCmod2XPB:4cYQHoQrvPiut3Sjs8fVR1BMZZpJjMTC4bsMTt9V71aQ
```

Note how the keys in the list are separated by colons.

Example config file snippet

```
"keyring": ["BnCsre9MPBeQK8QZBFznU2dJJ2GwtvnSMdemCmod2XPB",
            "4cYQHoQrvPiut3Sjs8fVR1BMZZpJjMTC4bsMTt9V71aQ"]
```

Default value (from a config file)

```
"keyring": []
```

6.1.3 database.host, database.port & database.name

The RethinkDB database hostname, port and name.

Example using environment variables

```
export BIGCHAINDB_DATABASE_HOST=localhost
export BIGCHAINDB_DATABASE_PORT=28015
export BIGCHAINDB_DATABASE_NAME=bigchain
```

Example config file snippet

```
"database": {
    "host": "localhost",
    "port": 28015,
    "name": "bigchain"
}
```

Default values (a snippet from `bigchaindb/__init__.py`)

```
'database': {
    'host': os.environ.get('BIGCHAINDB_DATABASE_HOST', 'localhost'),
    'port': 28015,
    'name': 'bigchain',
}
```

6.1.4 server.bind, server.workers & server.threads

These settings are for the Gunicorn HTTP server, which is used to serve the HTTP client-server API.

`server.bind` is where to bind the Gunicorn HTTP server socket. It's a string. It can be any valid value for Gunicorn's bind setting. If you want to allow IPv4 connections from anyone, on port 9984, use '0.0.0.0:9984'. In a production setting, we recommend you use Gunicorn behind a reverse proxy server. If Gunicorn and the reverse proxy are running on the same machine, then use 'localhost:PORT' where PORT is *not* 9984 (because the reverse proxy needs to listen on port 9984). Maybe use PORT=9983 in that case because we know 9983 isn't used. If Gunicorn and the reverse proxy are running on different machines, then use 'A.B.C.D:9984' where A.B.C.D is the IP address of the reverse proxy. There's more information about deploying behind a reverse proxy in the Gunicorn documentation. (They call it a proxy.)

`server.workers` is the number of worker processes for handling requests. If `None` (the default), the value will be (cpu_count * 2 + 1). `server.threads` is the number of threads-per-worker for handling requests. If `None` (the default), the value will be (cpu_count * 2 + 1). The HTTP server will be able to handle `server.workers` * `server.threads` requests simultaneously.

Example using environment variables

```
export BIGCHAINDB_SERVER_BIND=0.0.0.0:9984
export BIGCHAINDB_SERVER_WORKERS=5
export BIGCHAINDB_SERVER_THREADS=5
```

Example config file snippet

```
"server": {
    "bind": "0.0.0.0:9984",
    "workers": 5,
    "threads": 5
}
```

Default values (from a config file)

```
"server": {
    "bind": "localhost:9984",
    "workers": null,
    "threads": null
}
```

6.1.5 api_endpoint

`api_endpoint` is the URL where a BigchainDB client can get access to the HTTP client-server API.

Example using an environment variable

```
export BIGCHAINDB_API_ENDPOINT="http://localhost:9984/api/v1"
```

Example config file snippet

```
"api_endpoint": "http://webserver.blocks587.net:9984/api/v1"
```

Default value (from a config file)

```
"api_endpoint": "http://localhost:9984/api/v1"
```

6.1.6 statsd.host, statsd.port & statsd.rate

These settings are used to configure where, and how often, StatsD should send data for cluster monitoring purposes. `statsd.host` is the hostname of the monitoring server, where StatsD should send its data. `stats.port` is the port. `statsd.rate` is the fraction of transaction operations that should be sampled. It's a float between 0.0 and 1.0.

Example using environment variables

```
export BIGCHAINDB_STATSD_HOST="http://monitor.monitors-r-us.io"
export BIGCHAINDB_STATSD_PORT=8125
export BIGCHAINDB_STATSD_RATE=0.01
```

Example config file snippet: the default

```
"statsd": {"host": "localhost", "port": 8125, "rate": 0.01}
```

6.1.7 backlog_reassign_delay

Specifies how long, in seconds, transactions can remain in the backlog before being reassigned. Long-waiting transactions must be reassigned because the assigned node may no longer be responsive. The default duration is 120 seconds.

Example using environment variables

```
export BIGCHAINDB_BACKLOG_REASSIGN_DELAY=30
```

Default value (from a config file)

```
"backlog_reassign_delay": 120
```

6.2 Command Line Interface (CLI)

The command-line command to interact with BigchainDB Server is `bigchaindb`.

6.2.1 bigchaindb –help

Show help for the `bigchaindb` command. `bigchaindb -h` does the same thing.

6.2.2 bigchaindb –version

Show the version number. `bigchaindb -v` does the same thing.

6.2.3 bigchaindb configure

Generate a local config file (which can be used to set some or all BigchainDB node configuration settings). It will auto-generate a public-private keypair and then ask you for the values of other configuration settings. If you press Enter for a value, it will use the default value.

If you use the `-c` command-line option, it will generate the file at the specified path:

```
bigchaindb -c path/to/new_config.json configure
```

If you don't use the `-c` command-line option, the file will be written to `$HOME/.bigchaindb` (the default location where BigchainDB looks for a config file, if one isn't specified).

If you use the `-y` command-line option, then there won't be any interactive prompts: it will just generate a keypair and use the default values for all the other configuration settings.

```
bigchaindb -y configure
```

6.2.4 bigchaindb show-config

Show the values of the BigchainDB node configuration settings.

6.2.5 bigchaindb export-my-pubkey

Write the node's public key (i.e. one of its configuration values) to standard output (stdout).

6.2.6 bigchaindb init

Create a RethinkDB database, all RethinkDB database tables, various RethinkDB database indexes, and the genesis block.

Note: The `bigchaindb start` command (see below) always starts by trying a `bigchaindb init` first. If it sees that the RethinkDB database already exists, then it doesn't re-initialize the database. One doesn't have to do `bigchaindb init` before `bigchaindb start`. `bigchaindb init` is useful if you only want to initialize (but not start).

6.2.7 bigchaindb drop

Drop (erase) the RethinkDB database. You will be prompted to make sure. If you want to force-drop the database (i.e. skipping the yes/no prompt), then use `bigchaindb -y drop`

6.2.8 bigchaindb start

Start BigchainDB. It always begins by trying a `bigchaindb init` first. See the note in the documentation for `bigchaindb init`. You can also use the `--dev-start-rethinkdb` command line option to automatically start rethinkdb with bigchaindb if rethinkdb is not already running, e.g. `bigchaindb --dev-start-rethinkdb start`. Note that this will also shutdown rethinkdb when the bigchaindb process stops. The option `--dev-allow-temp-keypair` will generate a keypair on the fly if no keypair is found, this is useful when you want to run a temporary instance of BigchainDB in a Docker container, for example.

6.2.9 bigchaindb load

Write transactions to the backlog (for benchmarking tests). You can learn more about it using:

```
$ bigchaindb load -h
```

Note: This command uses the Python Server API to write transactions to the database. It *doesn't* use the HTTP API or a driver that wraps the HTTP API.

6.2.10 bigchaindb set-shards

Set the number of shards in the underlying datastore. For example, the following command will set the number of shards to four:

```
$ bigchaindb set-shards 4
```

6.2.11 bigchaindb set-replicas

Set the number of replicas (of each shard) in the underlying datastore. For example, the following command will set the number of replicas to three (i.e. it will set the replication factor to three):

```
$ bigchaindb set-replicas 3
```

CHAPTER 7

Drivers & Clients

Currently, the only language-native driver is written in the Python language.

We also provide the Transaction CLI to be able to script the building of transactions. You may be able to wrap this tool inside the language of your choice, and then use the HTTP API directly to post transactions.

7.1 The HTTP Client-Server API

Note: The HTTP client-server API is currently quite rudimentary. For example, there is no ability to do complex queries using the HTTP API. We plan to add querying capabilities in the future.

When you start Bigchaindb using *bigchaindb start*, an HTTP API is exposed at the address stored in the BigchainDB node configuration settings. The default is:

http://localhost:9984/api/v1/

but that address can be changed by changing the "API endpoint" configuration setting (e.g. in a local config file). There's more information about setting the API endpoint in *the section about BigchainDB Configuration Settings*.

There are other configuration settings related to the web server (serving the HTTP API). In particular, the default is for the web server socket to bind to `localhost:9984` but that can be changed (e.g. to `0.0.0.0:9984`). For more details, see the "server" settings ("bind", "workers" and "threads") in *the section about BigchainDB Configuration Settings*.

7.1.1 API Root

If you send an HTTP GET request to e.g. `http://localhost:9984` (with no `/api/v1/` on the end), then you should get an HTTP response with something like the following in the body:

```
{
  "api_endpoint": "http://localhost:9984/api/v1",
```

```
  "keyring": [
    "6qHyZew94NMmUTYyHnkZsB8cxJYuRNEiEpXHe1ih9QX3",
    "AdDuyrTyjrDt935YnFu4VBCVDhHtY2Y6rcy7x2TFeiRi"
  ],
  "public_key": "AiygKSRhZWTxxYT4AfgKoTG4TZAoPsWoEt6C6bLq4jJR",
  "software": "BigchainDB",
  "version": "0.6.0"
}
```

7.1.2 POST /transactions/

POST /transactions/

Push a new transaction.

Note: The posted transaction should be a valid and signed transaction. The steps to build a valid transaction are beyond the scope of this page. One would normally use a driver such as the BigchainDB Python Driver to build a valid transaction. The exact contents of a valid transaction depend on the associated public/private keypairs.

Example request:

```
POST /transactions/ HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "transaction": {
    "conditions": [
      {
        "cid": 0,
        "condition": {
          "uri": "cc:4:20:fSlVCKNSzSl0meiwwuUk5JpJ0KLlECTqbd25KyQefFY:96",
          "details": {
            "signature": null,
            "type": "fulfillment",
            "type_id": 4,
            "bitmask": 32,
            "public_key": "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
          }
        },
        "amount": 1,
        "owners_after": [
          "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
        ]
      }
    ],
    "operation": "CREATE",
    "asset": {
      "divisible": false,
      "updatable": false,
      "data": null,
      "id": "b57801f8-b865-4360-9d1a-3e3009f5ce01",
      "refillable": false
    },
    "metadata": null,
    "fulfillments": [
      {
        "fid": 0,
```

```
        "input": null,
        "fulfillment":
↪"cf:4:fSlVCKNSzSl0meiwwuUk5JpJ0KLlECTqbd25KyQefFaf8bQVH1gesZGEGZepCE8_kgo-
↪UfBrCHPlvBsnAsfq56GWjrLTyZ9NXISwcyJ3zmygnVhCMG8xzE6c9fj1-6wK",
        "owners_before": [
            "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
        ]
      }
    ]
  },
  "id": "65f1f69b6ebf995a7b2c5ae8a6fb480ce20f0e8f1eb1d77d75f37ab00ccdeec3",
  "version": 1
}
```

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": "65f1f69b6ebf995a7b2c5ae8a6fb480ce20f0e8f1eb1d77d75f37ab00ccdeec3",
  "version": 1,
  "transaction": {
    "conditions": [
      {
        "amount": 1,
        "condition": {
          "uri": "cc:4:20:fSlVCKNSzSl0meiwwuUk5JpJ0KLlECTqbd25KyQefFY:96",
          "details": {
            "signature": null,
            "type_id": 4,
            "type": "fulfillment",
            "bitmask": 32,
            "public_key": "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
          }
        },
        "owners_after": [
          "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
        ],
        "cid": 0
      }
    ],
    "fulfillments": [
      {
        "input": null,
        "fulfillment":
↪"cf:4:fSlVCKNSzSl0meiwwuUk5JpJ0KLlECTqbd25KyQefFaf8bQVH1gesZGEGZepCE8_kgo-
↪UfBrCHPlvBsnAsfq56GWjrLTyZ9NXISwcyJ3zmygnVhCMG8xzE6c9fj1-6wK",
        "fid": 0,
        "owners_before": [
          "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
        ]
      }
    ],
    "operation": "CREATE",
    "asset": {
      "updatable": false,
      "refillable": false,
```

```
      "divisible": false,
      "data": null,
      "id": "b57801f8-b865-4360-9d1a-3e3009f5ce01"
    },
    "metadata": null
  }
}
```

Status Codes

- 201 Created – A new transaction was created.
- 400 Bad Request – The transaction was invalid and not created.

7.1.3 GET /transactions/{tx_id}/status

GET /transactions/{tx_id}/status

Get the status of the transaction with the ID `tx_id`, if a transaction with that `tx_id` exists.

The possible status values are `backlog`, `undecided`, `valid` or `invalid`.

Parameters

- **tx_id** (*hex string*) – transaction ID

Example request:

```
GET /transactions/
↪65f1f69b6ebf995a7b2c5ae8a6fb480ce20f0e8f1eb1d77d75f37ab00ccdeec3/status HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "valid"
}
```

Status Codes

- 200 OK – A transaction with that ID was found and the status is returned.
- 404 Not Found – A transaction with that ID was not found.

7.1.4 GET /transactions/{tx_id}

GET /transactions/{tx_id}

Get the transaction with the ID `tx_id`.

This endpoint returns only a transaction from a `VALID` or `UNDECIDED` block on `bigchain`, if exists.

Parameters

- **tx_id** (*hex string*) – transaction ID

Example request:

```
GET /transactions/
↪65f1f69b6ebf995a7b2c5ae8a6fb480ce20f0e8f1eb1d77d75f37ab00ccdeec3 HTTP/1.1
Host: example.com
```

Example response:


```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "transaction": {
    "conditions": [
      {
        "cid": 0,
        "condition": {
          "uri": "cc:4:20:fSlVCKNSzSl0meiwwuUk5JpJ0KLlECTqbd25KyQefFY:96",
          "details": {
            "signature": null,
            "type": "fulfillment",
            "type_id": 4,
            "bitmask": 32,
            "public_key": "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
          }
        },
        "amount": 1,
        "owners_after": [
          "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
        ]
      }
    ],
    "operation": "CREATE",
    "asset": {
      "divisible": false,
      "updatable": false,
      "data": null,
      "id": "b57801f8-b865-4360-9d1a-3e3009f5ce01",
      "refillable": false
    },
    "metadata": null,
    "fulfillments": [
      {
        "fid": 0,
        "input": null,
        "fulfillment":
↪"cf:4:fSlVCKNSzSl0meiwwuUk5JpJ0KLlECTqbd25KyQefFaf8bQVH1gesZGEGZepCE8_kgo-
↪UfBrCHPlvBsnAsfq56GWjrLTyZ9NXISwcyJ3zmygnVhCMG8xzE6c9fj1-6wK",
        "owners_before": [
          "9RaWxppkP9UyYWA7NJb5FcgkzfJNPfvPX3FCNw2T5Pwb"
        ]
      }
    ]
  },
  "id": "65f1f69b6ebf995a7b2c5ae8a6fb480ce20f0e8f1eb1d77d75f37ab00ccdeec3",
  "version": 1
}
```


Status Codes

- 200 OK – A transaction with that ID was found.
- 404 Not Found – A transaction with that ID was not found.

7.1.5 GET /unspents/

Note: This endpoint (unspents) is not yet implemented. We published it here for preview and comment.

GET /unspents?owner_after={owner_after}

Get a list of links to transactions' conditions that have not been used in a previous transaction and could hence be called unspent conditions/outputs (or simply: unspents).

This endpoint will return a `HTTP 400 Bad Request` if the querystring `owner_after` happens to not be defined in the request.

Note that if unspents for a certain `owner_after` have not been found by the server, this will result in the server returning a 200 OK HTTP status code and an empty list in the response's body.

Parameters

- **owner_after** (*base58 encoded string*) – A public key, able to validly spend an output of a transaction, assuming the user also has the corresponding private key.

Example request:

```
GET /unspents?owner_after=1AAAbbb...ccc HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  '../transactions/
↪2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e/conditions/0',
  '../transactions/
↪2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e/conditions/1'
]
```

Status Codes

- 200 OK – A list of outputs were found and returned in the body of the response.
- 400 Bad Request – The request wasn't understood by the server, e.g. the `owner_after` querystring was not included in the request.

7.2 Example Apps

Warning: There are some example BigchainDB apps (i.e. apps which use BigchainDB) in the GitHub repository named bigchaindb-examples. They were created before there was much of an HTTP API, so instead of communicating with a BigchainDB node via the HTTP API, they communicate directly with the node using the BigchainDB

Python server API and the RethinkDB Python Driver. That's not how a real production app would work. The HTTP API is getting better, and we recommend using it to communicate with BigchainDB nodes.

Moreover, because of changes to the BigchainDB Server code, some of the examples in the bigchaindb-examples repo might not work anymore, or they might not work as expected.

In the future, we hope to create a set of examples using the HTTP API (or wrappers of it, such as the Python Driver API).

CHAPTER 8

Clusters & Federations

8.1 Set Up a Federation

This section is about how to set up a BigchainDB *federation*, where each node is operated by a different operator. If you want to set up and run a testing cluster on AWS (where all nodes are operated by you), then see the section about that.

8.1.1 Initial Checklist

- Do you have a governance process for making federation-level decisions, such as how to admit new members?
- What will you store in creation transactions (data payload)? Is there a data schema?
- Will you use transfer transactions? Will they include a non-empty data payload?
- Who will be allowed to submit transactions? Who will be allowed to read or query transactions? How will you enforce the access rules?

8.1.2 Set Up the Initial Cluster

The federation must decide some things before setting up the initial cluster (initial set of BigchainDB nodes):

1. Who will operate a node in the initial cluster?
2. What will the replication factor be? (It must be 3 or more for RethinkDB failover to work.)
3. Which node will be responsible for sending the commands to configure the RethinkDB database?

Once those things have been decided, each node operator can begin setting up their BigchainDB (production) node.

Each node operator will eventually need two pieces of information from all other nodes in the federation:

1. Their RethinkDB hostname, e.g. `rdb.farm2.organization.org`
2. Their BigchainDB public key, e.g. `Eky3nkbxDTMgkmiJC8i5hKyVFiAQNmPP4a2G4JdDxJCK`

8.2 Backing Up & Restoring Data

There are several ways to backup and restore the data in a BigchainDB cluster.

8.2.1 RethinkDB's Replication as a form of Backup

RethinkDB already has internal replication: every document is stored on *R* different nodes, where *R* is the replication factor (set using `bigchaindb set-replicas R`). Those replicas can be thought of as "live backups" because if one node goes down, the cluster will continue to work and no data will be lost.

At this point, there should be someone saying, "But replication isn't backup¡'

It's true. Replication alone isn't enough, because something bad might happen *inside* the database, and that could affect the replicas. For example, what if someone logged in as a RethinkDB admin and did a "drop table"? We currently plan for each node to be protected by a next-generation firewall (or something similar) to prevent such things from getting very far. For example, see issue #240.

Nevertheless, you should still consider having normal, "cold" backups, because bad things can still happen.

8.2.2 Live Replication of RethinkDB Data Files

Each BigchainDB node stores its subset of the RethinkDB data in one directory. You could set up the node's file system so that directory lives on its own hard drive. Furthermore, you could make that hard drive part of a RAID array, so that a second hard drive would always have a copy of the original. If the original hard drive fails, then the second hard drive could take its place and the node would continue to function. Meanwhile, the original hard drive could be replaced.

That's just one possible way of setting up the file system so as to provide extra reliability.

Another way to get similar reliability would be to mount the RethinkDB data directory on an Amazon EBS volume. Each Amazon EBS volume is, "automatically replicated within its Availability Zone to protect you from component failure, offering high availability and durability."

See the section on setting up storage for RethinkDB for more details.

As with shard replication, live file-system replication protects against many failure modes, but it doesn't protect against them all. You should still consider having normal, "cold" backups.

8.2.3 rethinkdb dump (to a File)

RethinkDB can create an archive of all data in the cluster (or all data in specified tables), as a compressed file. According to the RethinkDB blog post when that functionality became available:

> Since the backup process is using client drivers, it automatically takes advantage of the MVCC [multiversion concurrency control] functionality built into RethinkDB. It will use some cluster resources, but will not lock out any of the clients, so you can safely run it on a live cluster.

To back up all the data in a BigchainDB cluster, the RethinkDB admin user must run a command like the following on one of the nodes:

```
rethinkdb dump -e bigchain.bigchain -e bigchain.votes
```

That should write a file named `rethinkdb_dump_<date>_<time>.tar.gz`. The `-e` option is used to specify which tables should be exported. You probably don't need to export the backlog table, but you definitely need to export the bigchain and votes tables. `bigchain.votes` means the `votes` table in the RethinkDB database named `bigchain`. It's possible that your database has a different name: the database name is a BigchainDB configuration setting. The default name is `bigchain`. (Tip: you can see the values of all configuration settings using the `bigchaindb show-config` command.)

There's more information about the `rethinkdb dump` command in the RethinkDB documentation. It also explains how to restore data to a cluster from an archive file.

Notes

- If the `rethinkdb dump` subcommand fails and the last line of the Traceback says "NameError: name 'file' is not defined", then you need to update your RethinkDB Python driver; do a `pip install --upgrade rethinkdb`
- It might take a long time to backup data this way. The more data, the longer it will take.
- You need enough free disk space to store the backup file.
- If a document changes after the backup starts but before it ends, then the changed document may not be in the final backup. This shouldn't be a problem for BigchainDB, because blocks and votes can't change anyway.
- `rethinkdb dump` saves data and secondary indexes, but does *not* save cluster metadata. You will need to recreate your cluster setup yourself after you run `rethinkdb restore`.
- RethinkDB also has subcommands to import/export collections of JSON or CSV files. While one could use those for backup/restore, it wouldn't be very practical.

8.2.4 Client-Side Backup

In the future, it will be possible for clients to query for the blocks containing the transactions they care about, and for the votes on those blocks. They could save a local copy of those blocks and votes.

How could we be sure blocks and votes from a client are valid?

All blocks and votes are signed by federation nodes. Only federation nodes can produce valid signatures because only federation nodes have the necessary private keys. A client can't produce a valid signature for a block or vote.

Could we restore an entire BigchainDB database using client-saved blocks and votes?

Yes, in principle, but it would be difficult to know if you've recovered every block and vote. Votes link to the block they're voting on and to the previous block, so one could detect some missing blocks. It would be difficult to know if you've recovered all the votes.

8.2.5 Backup by Copying RethinkDB Data Files

It's *possible* to back up a BigchainDB database by creating a point-in-time copy of the RethinkDB data files (on all nodes, at roughly the same time). It's not a very practical approach to backup: the resulting set of files will be much larger (collectively) than what one would get using `rethinkdb dump`, and there are no guarantees on how consistent that data will be, especially for recently-written data.

If you're curious about what's involved, see the MongoDB documentation about "Backup by Copying Underlying Data Files". (Yes, that's documentation for MongoDB, but the principles are the same.)

See the last subsection of this page for a better way to use this idea.

8.2.6 Incremental or Continuous Backup

Incremental backup is where backup happens on a regular basis (e.g. daily), and each one only records the changes since the last backup.

Continuous backup might mean incremental backup on a very regular basis (e.g. every ten minutes), or it might mean backup of every database operation as it happens. The latter is also called transaction logging or continuous archiving.

At the time of writing, RethinkDB didn't have a built-in incremental or continuous backup capability, but the idea was raised in RethinkDB issues #89 and #5890. On July 5, 2016, Daniel Mewes (of RethinkDB) wrote the following comment on issue #5890: "We would like to add this feature [continuous backup], but haven't started working on it yet."

To get a sense of what continuous backup might look like for RethinkDB, one can look at the continuous backup options available for MongoDB. MongoDB, the company, offers continuous backup with Ops Manager (self-hosted) or Cloud Manager (fully managed). Features include:

- It "continuously maintains backups, so if your MongoDB deployment experiences a failure, the most recent backup is only moments behind..."
- It "offers point-in-time backups of replica sets and cluster-wide snapshots of sharded clusters. You can restore to precisely the moment you need, quickly and safely."
- "You can rebuild entire running clusters, just from your backups."
- It enables, "fast and seamless provisioning of new dev and test environments."

The MongoDB documentation has more details about how Ops Manager Backup works.

Considerations for BigchainDB:

- We'd like the cost of backup to be low. To get a sense of the cost, MongoDB Cloud Manager backup costed $30 / GB / year prepaid. One thousand gigabytes backed up (i.e. about a terabyte) would cost 30 thousand US dollars per year. (That's just for the backup; there's also a cost per server per year.)
- We'd like the backup to be decentralized, with no single point of control or single point of failure. (Note: some file systems have a single point of failure. For example, HDFS has one Namenode.)
- We only care to back up blocks and votes, and once written, those never change. There are no updates or deletes, just new blocks and votes.

8.2.7 Combining RethinkDB Replication with Storage Snapshots

Although it's not advertised as such, RethinkDB's built-in replication feature is similar to continous backup, except the "backup" (i.e. the set of replica shards) is spread across all the nodes. One could take that idea a bit farther by creating a set of backup-only servers with one full backup:

- Give all the original BigchainDB nodes (RethinkDB nodes) the server tag `original`. This is the default if you used the RethinkDB config file suggested in the section titled Configure RethinkDB Server.
- Set up a group of servers running RethinkDB only, and give them the server tag `backup`. The `backup` servers could be geographically separated from all the `original` nodes (or not; it's up to the federation).
- Clients shouldn't be able to read from or write to servers in the `backup` set.
- Send a RethinkDB reconfigure command to the RethinkDB cluster to make it so that the `original` set has the same number of replicas as before (or maybe one less), and the `backup` set has one replica. Also, make sure the `primary_replica_tag='original'` so that all primary shards live on the `original` nodes.

The RethinkDB documentation on sharding and replication has the details of how to set server tags and do RethinkDB reconfiguration.

Once you've set up a set of backup-only RethinkDB servers, you could make a point-in-time snapshot of their storage devices, as a form of backup.

You might want to disconnect the `backup` set from the `original` set first, and then wait for reads and writes in the `backup` set to stop. (The `backup` set should have only one copy of each shard, so there's no opportunity for inconsistency between shards of the `backup` set.)

You will want to re-connect the `backup` set to the `original` set as soon as possible, so it's able to catch up.

If something bad happens to the entire original BigchainDB cluster (including the `backup` set) and you need to restore it from a snapshot, you can, but before you make BigchainDB live, you should 1) delete all entries in the backlog table, 2) delete all blocks after the last voted-valid block, 3) delete all votes on the blocks deleted in part 2, and 4) rebuild the RethinkDB indexes.

NOTE: Sometimes snapshots are *incremental*. For example, Amazon EBS snapshots are incremental, meaning "only the blocks on the device that have changed after your most recent snapshot are saved. **This minimizes the time required to create the snapshot and saves on storage costs.**" [Emphasis added]

8.3 Deploy a Testing Cluster on AWS

This section explains a way to deploy a cluster of BigchainDB nodes on Amazon Web Services (AWS) for testing purposes.

8.3.1 Why?

Why would anyone want to deploy a centrally-controlled BigchainDB cluster? Isn't BigchainDB supposed to be decentralized, where each node is controlled by a different person or organization?

Yes! These scripts are for deploying a testing cluster, not a production cluster.

8.3.2 How?

We use some Bash and Python scripts to launch several instances (virtual servers) on Amazon Elastic Compute Cloud (EC2). Then we use Fabric to install RethinkDB and BigchainDB on all those instances.

8.3.3 Python Setup

The instructions that follow have been tested on Ubuntu 14.04, but may also work on similar distros or operating systems.

Note: Our Python scripts for deploying to AWS use Python 2 because Fabric doesn't work with Python 3.

Maybe create a Python 2 virtual environment and activate it. Then install the following Python packages (in that virtual environment):

```
pip install fabric fabtools requests boto3 awscli
```

What did you just install?

- "Fabric is a Python (2.5-2.7) library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks."
- fabtools are "tools for writing awesome Fabric files"
- requests is a Python package/library for sending HTTP requests
- "Boto is the Amazon Web Services (AWS) SDK for Python, which allows Python developers to write software that makes use of Amazon services like S3 and EC2." (`boto3` is the name of the latest Boto package.)

- The aws-cli package, which is an AWS Command Line Interface (CLI).

8.3.4 Setting up in AWS

See the page about basic AWS Setup in the Appendices.

8.3.5 Get Enough Amazon Elastic IP Addresses

The AWS cluster deployment scripts use elastic IP addresses (although that may change in the future). By default, AWS accounts get five elastic IP addresses. If you want to deploy a cluster with more than five nodes, then you will need more than five elastic IP addresses; you may have to apply for those; see the AWS documentation on elastic IP addresses.

8.3.6 Create an Amazon EC2 Security Group

Go to the AWS EC2 Console and select "Security Groups" in the left sidebar. Click the "Create Security Group" button. You can name it whatever you like. (Notes: The default name in the example AWS deployment configuration file is `bigchaindb`. We had problems with names containing dashes.) The description should be something to help you remember what the security group is for.

For a super lax, somewhat risky, anything-can-enter security group, add these rules for Inbound traffic:

- Type = All TCP, Protocol = TCP, Port Range = 0-65535, Source = 0.0.0.0/0
- Type = SSH, Protocol = SSH, Port Range = 22, Source = 0.0.0.0/0
- Type = All UDP, Protocol = UDP, Port Range = 0-65535, Source = 0.0.0.0/0
- Type = All ICMP, Protocol = ICMP, Port Range = 0-65535, Source = 0.0.0.0/0

(Note: Source = 0.0.0.0/0 is CIDR notation for "allow this traffic to come from *any* IP address.")

If you want to set up a more secure security group, see the Notes for Firewall Setup.

8.3.7 Deploy a BigchainDB Monitor

This step is optional.

One way to monitor a BigchainDB cluster is to use the monitoring setup described in the Monitoring section of this documentation. If you want to do that, then you may want to deploy the monitoring server first, so you can tell your BigchainDB nodes where to send their monitoring data.

You can deploy a monitoring server on AWS. To do that, go to the AWS EC2 Console and launch an instance:

1. Choose an AMI: select Ubuntu Server 14.04 LTS.
2. Choose an Instance Type: a t2.micro will suffice.
3. Configure Instance Details: you can accept the defaults, but feel free to change them.
4. Add Storage: A "Root" volume type should already be included. You *could* store monitoring data there (e.g. in a folder named `/influxdb-data`) but we will attach another volume and store the monitoring data there instead. Select "Add New Volume" and an EBS volume type.
5. Tag Instance: give your instance a memorable name.
6. Configure Security Group: choose your bigchaindb security group.

7. Review and launch your instance.

When it asks, choose an existing key pair: the one you created earlier (named `bigchaindb`).

Give your instance some time to launch and become able to accept SSH connections. You can see its current status in the AWS EC2 Console (in the "Instances" section). SSH into your instance using something like:

```
cd deploy-cluster-aws
ssh -i pem/bigchaindb.pem ubuntu@ec2-52-58-157-229.eu-central-1.compute.amazonaws.com
```

where `ec2-52-58-157-229.eu-central-1.compute.amazonaws.com` should be replaced by your new instance's EC2 hostname. (To get that, go to the AWS EC2 Console, select Instances, click on your newly-launched instance, and copy its "Public DNS" name.)

Next, create a file system on the attached volume, make a directory named `/influxdb-data`, and set the attached volume's mount point to be `/influxdb-data`. For detailed instructions on how to do that, see the AWS documentation for Making an Amazon EBS Volume Available for Use.

Then install Docker and Docker Compose:

```
# in a Python 2.5-2.7 virtual environment where fabric, boto3, etc. are installed
fab --fabfile=fabfile-monitor.py --hosts=<EC2 hostname> install_docker
```

After Docker is installed, we can run the monitor with:

```
fab --fabfile=fabfile-monitor.py --hosts=<EC2 hostname> run_monitor
```

For more information about monitoring (e.g. how to view the Grafana dashboard in your web browser), see the Monitoring section of this documentation.

To configure a BigchainDB node to send monitoring data to the monitoring server, change the statsd host in the configuration of the BigchainDB node. The section on Configuring a BigchainDB Node explains how you can do that. (For example, you can change the statsd host in `$HOME/.bigchaindb`.)

8.3.8 Deploy a BigchainDB Cluster

Step 1

Suppose *N* is the number of nodes you want in your BigchainDB cluster. If you already have a set of *N* BigchainDB configuration files in the `deploy-cluster-aws/confiles` directory, then you can jump to the next step. To create such a set, you can do something like:

```
# in a Python 3 virtual environment where bigchaindb is installed
cd bigchaindb
cd deploy-cluster-aws
./make_confiles.sh confiles 3
```

That will create three (3) *default* BigchainDB configuration files in the `deploy-cluster-aws/confiles` directory (which will be created if it doesn't already exist). The three files will be named `bcdb_conf0`, `bcdb_conf1`, and `bcdb_conf2`.

You can look inside those files if you're curious. For example, the default keyring is an empty list. Later, the deployment script automatically changes the keyring of each node to be a list of the public keys of all other nodes. Other changes are also made. That is, the configuration files generated in this step are *not* what will be sent to the deployed nodes; they're just a starting point.

Step 2

Step 2 is to make an AWS deployment configuration file, if necessary. There's an example AWS configuration file named `example_deploy_conf.py`. It has many comments explaining each setting. The settings in that file are (or should be):

```
NUM_NODES=3
BRANCH="master"
WHAT_TO_DEPLOY="servers"
SSH_KEY_NAME="not-set-yet"
USE_KEYPAIRS_FILE=False
IMAGE_ID="ami-8504fdea"
INSTANCE_TYPE="t2.medium"
SECURITY_GROUP="bigchaindb"
USING_EBS=True
EBS_VOLUME_SIZE=30
EBS_OPTIMIZED=False
BIND_HTTP_TO_LOCALHOST=True
```

Make a copy of that file and call it whatever you like (e.g. `cp example_deploy_conf.py my_deploy_conf.py`). You can leave most of the settings at their default values, but you must change the value of `SSH_KEY_NAME` to the name of your private SSH key. You can do that with a text editor. Set `SSH_KEY_NAME` to the name you used for `<key-name>` when you generated an RSA key pair for SSH (in basic AWS setup).

You'll also want to change the `IMAGE_ID` to one that's up-to-date and available in your AWS region. If you don't remember your AWS region, then look in your `$HOME/.aws/config` file. You can find an up-to-date Ubuntu image ID for your region at https://cloud-images.ubuntu.com/locator/ec2/. An example search string is "eu-central-1 16.04 LTS amd64 hvm:ebs-ssd". You should replace "eu-central-1" with your region name.

If you want your nodes to have a predictable set of pre-generated keypairs, then you should 1) set `USE_KEYPAIRS_FILE=True` in the AWS deployment configuration file, and 2) provide a `keypairs.py` file containing enough keypairs for all of your nodes. You can generate a `keypairs.py` file using the `write_keypairs_file.py` script. For example:

```
# in a Python 3 virtual environment where bigchaindb is installed
cd bigchaindb
cd deploy-cluster-aws
python3 write_keypairs_file.py 100
```

The above command generates a `keypairs.py` file with 100 keypairs. You can generate more keypairs than you need, so you can use the same list over and over again, for different numbers of servers. The deployment scripts will only use the first NUM_NODES keypairs.

Step 3

Step 3 is to launch the nodes ("instances") on AWS, to install all the necessary software on them, configure the software, run the software, and more. Here's how you'd do that:

```
# in a Python 2.5-2.7 virtual environment where fabric, boto3, etc. are installed
cd bigchaindb
cd deploy-cluster-aws
./awsdeploy.sh my_deploy_conf.py
# Only if you want to set the replication factor to 3
fab set_replicas:3
# Only if you want to start BigchainDB on all the nodes:
fab start_bigchaindb
```

`awsdeploy.sh` is a Bash script which calls some Python and Fabric scripts. If you're curious what it does, the source code has many explanatory comments.

It should take a few minutes for the deployment to finish. If you run into problems, see the section on **Known Deployment Issues** below.

The EC2 Console has a section where you can see all the instances you have running on EC2. You can `ssh` into a running instance using a command like:

```
ssh -i pem/bigchaindb.pem ubuntu@ec2-52-29-197-211.eu-central-1.compute.amazonaws.com
```

except you'd replace the `ec2-52-29-197-211.eu-central-1.compute.amazonaws.com` with the public DNS name of the instance you want to `ssh` into. You can get that from the EC2 Console: just click on an instance and look in its details pane at the bottom of the screen. Some commands you might try:

```
ip addr show
sudo service rethinkdb status
bigchaindb --help
bigchaindb show-config
```

You can also check out the RethinkDB web interface. The way to do that depends on how `BIND_HTTP_TO_LOCALHOST` was set in your AWS deployment configuration file:

- If it was set to `False`, then just go to your web browser and visit a web address like `http://ec2-52-29-197-211.eu-central-1.compute.amazonaws.com:8080/`. (Replace `ec2-...aws.com` with the hostname of one of your instances.)
- If it was set to `True` (the default in the example config file), then follow the instructions in the "Via a SOCKS proxy" section of the "Secure your cluster" page of the RethinkDB documentation.

8.3.9 Server Monitoring with New Relic

New Relic is a business that provides several monitoring services. One of those services, called Server Monitoring, can be used to monitor things like CPU usage and Network I/O on BigchainDB instances. To do that:

1. Sign up for a New Relic account
2. Get your New Relic license key
3. Put that key in an environment variable named `NEWRELIC_KEY`. For example, you might add a line like the following to your `~/.bashrc` file (if you use Bash): `export NEWRELIC_KEY=<insert your key here>`
4. Once you've deployed a BigchainDB cluster on AWS as above, you can install a New Relic system monitor (agent) on all the instances using:

```
# in a Python 2.5-2.7 virtual environment where fabric, boto3, etc. are installed
fab install_newrelic
```

Once the New Relic system monitor (agent) is installed on the instances, it will start sending server stats to New Relic on a regular basis. It may take a few minutes for data to show up in your New Relic dashboard (under New Relic Servers).

8.3.10 Shutting Down a Cluster

There are fees associated with running instances on EC2, so if you're not using them, you should terminate them. You can do that using the AWS EC2 Console.

The same is true of your allocated elastic IP addresses. There's a small fee to keep them allocated if they're not associated with a running instance. You can release them using the AWS EC2 Console, or by using a handy little script named `release_eips.py`. For example:

```
$ python release_eips.py
You have 2 allocated elactic IPs which are not associated with instances
0: Releasing 52.58.110.110
(It has Domain = vpc.)
1: Releasing 52.58.107.211
(It has Domain = vpc.)
```

8.3.11 Known Deployment Issues

NetworkError

If you tested with a high sequence it might be possible that you run into an error message like this:

```
NetworkError: Host key for ec2-xx-xx-xx-xx.eu-central-1.compute.amazonaws.com
did not match pre-existing key! Server's key was changed recently, or possible
man-in-the-middle attack.
```

If so, just clean up your `known_hosts` file and start again. For example, you might copy your current `known_hosts` file to `old_known_hosts` like so:

```
mv ~/.ssh/known_hosts ~/.ssh/old_known_hosts
```

Then terminate your instances and try deploying again with a different tag.

Failure of sudo apt-get update

The first thing that's done on all the instances, once they're running, is basically `sudo apt-get update`. Sometimes that fails. If so, just terminate your instances and try deploying again with a different tag. (These problems seem to be time-bounded, so maybe wait a couple of hours before retrying.)

Failure when Installing Base Software

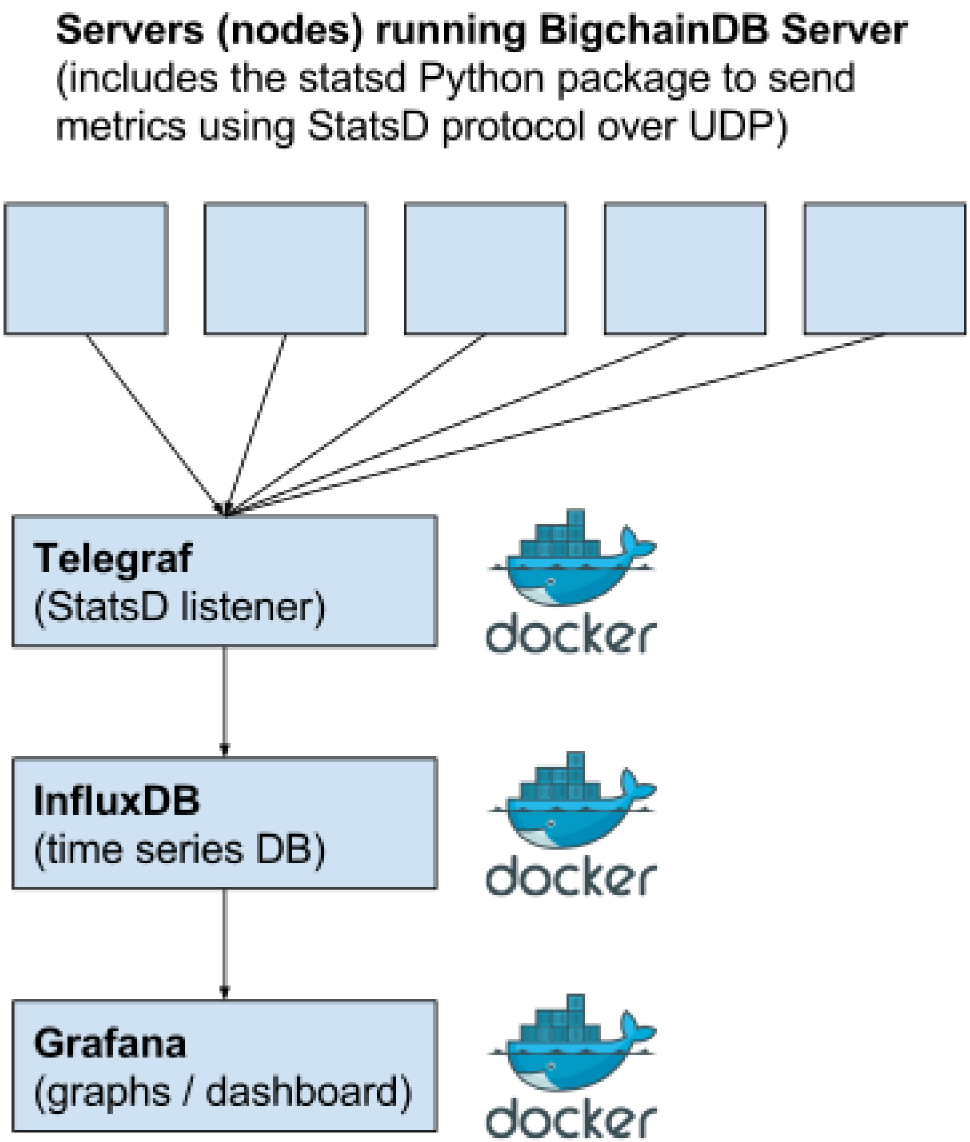
If you get an error with installing the base software on the instances, then just terminate your instances and try deploying again with a different tag.

8.4 Cluster Monitoring

BigchainDB uses StatsD for cluster monitoring. We require some additional infrastructure to take full advantage of its functionality:

- an agent to listen for metrics: Telegraf,
- a time-series database: InfluxDB, and
- a frontend to display analytics: Grafana.

We put each of those inside its own Docker container. The whole system is illustrated below.



For ease of use, we've created a Docker *Compose file* (named `docker-compose-monitor.yml`) to define the monitoring system setup. To use it, just go to to the top `bigchaindb` directory and run:

```
$ docker-compose -f docker-compose-monitor.yml build
$ docker-compose -f docker-compose-monitor.yml up
```

It is also possible to mount a host directory as a data volume for InfluxDB by setting the `INFLUXDB_DATA` environment variable:

```
$ INFLUXDB_DATA=/data docker-compose -f docker-compose-monitor.yml up
```

You can view the Grafana dashboard in your web browser at:

http://localhost:3000/dashboard/script/bigchaindb_dashboard.js

(You may want to replace `localhost` with another hostname in that URL, e.g. the hostname of a remote monitoring server.)

The login and password are `admin` by default. If BigchainDB is running and processing transactions, you should see analytics—if not, start BigchainDB and load some test transactions:

```
$ bigchaindb load
```

then refresh the page after a few seconds.

If you're not interested in monitoring, don't worry: BigchainDB will function just fine without any monitoring setup.

Feel free to modify the custom Grafana dashboard to your liking!

CHAPTER 9

Topic Guides

Note: Most of the Topic Guides have been moved over to the root BigchainDB project docs.

9.1 The Transaction, Block and Vote Models

This page about transaction concepts and data models was getting too big, so it was split into smaller pages. It will be deleted eventually, so update your links. Here's where you can find the new pages:

- Transaction Concepts
- Data Models (all of them)

CHAPTER 10

Data Models

BigchainDB stores all data in the underlying database as JSON documents (conceptually, at least). There are three main kinds:

1. Transactions, which contain digital assets, conditions, fulfillments and other things
2. Blocks
3. Votes

This section unpacks each one in turn.

10.1 The Transaction Model

A transaction has the following structure:

```
{
    "id": "<hash of transaction, excluding signatures (see explanation)>",
    "version": "<version number of the transaction model>",
    "transaction": {
        "fulfillments": ["<list of fulfillments>"],
        "conditions": ["<list of conditions>"],
        "operation": "<string>",
        "asset": "<digital asset description (explained in the next section)>",
        "metadata": {
            "id": "<uuid>",
            "data": "<any JSON document>"
        }
    }
}
```

Here's some explanation of the contents of a *transaction*:

- *id*: The id of the transaction, and also the database primary key.
- *version*: Version number of the transaction model, so that software can support different transaction models.

- ***transaction***:
  - **fulfillments**: List of fulfillments. Each *fulfillment* contains a pointer to an unspent asset and a *crypto fulfillment* that satisfies a spending condition set on the unspent asset. A *fulfillment* is usually a signature proving the ownership of the asset. See *Crypto-Conditions and Fulfillments*.
  - **conditions**: List of conditions. Each *condition* is a *crypto-condition* that needs to be fulfilled by a transfer transaction in order to transfer ownership to new owners. See *Crypto-Conditions and Fulfillments*.
  - **operation**: String representation of the *operation* being performed (currently either "CREATE", "TRANSFER" or "GENESIS"). It determines how the transaction should be validated.
    - **asset**: Definition of the digital *asset*. See next section.
  - **metadata:**
    - *id*: UUID version 4 (random) converted to a string of hex digits in standard form.
    - *data*: Can be any JSON document. It may be empty in the case of a transfer transaction.

Later, when we get to the models for the block and the vote, we'll see that both include a signature (from the node which created it). You may wonder why transactions don't have signatures... The answer is that they do! They're just hidden inside the `fulfillment` string of each fulfillment. A creation transaction is signed by whoever created it. A transfer transaction is signed by whoever currently controls or owns it.

What gets signed? For each fulfillment in the transaction, the "fullfillment message" that gets signed includes the `operation`, `data`, `version`, `id`, corresponding `condition`, and the fulfillment itself, except with its fulfillment string set to `null`. The computed signature goes into creating the `fulfillment` string of the fulfillment.

10.2 The Digital Asset Model

To avoid redundant data in transactions, the digital asset model is different for `CREATE` and `TRANSFER` transactions.

A digital asset's properties are defined in a `CREATE` transaction with the following model:

```
{
    "id": "<uuid>",
    "divisible": "<true | false>",
    "updatable": "<true | false>",
    "refillable": "<true | false>",
    "data": "<json document>"
}
```

For `TRANSFER` transactions we only keep the asset id.

```
{
    "id": "<uuid>"
}
```

- `id`: UUID version 4 (random) converted to a string of hex digits in standard form. Added server side.
- `divisible`: Whether the asset is divisible or not. Defaults to false.
- `updatable`: Whether the data in the asset can be updated in the future or not. Defaults to false.
- `refillable`: Whether the amount of the asset can change after its creation. Defaults to false.
- `data`: A user supplied JSON document with custom information about the asset. Defaults to null.

- *amount*: The amount of "shares". Only relevant if the asset is marked as divisible. Defaults to 1. The amount is not specified in the asset, but in the conditions (see next section).

At the time of this writing, updatable and refillable assets are not yet implemented.

10.3 Crypto-Conditions and Fulfillments

To create a transaction that transfers an asset to new owners, one must fulfill the asset's current conditions (crypto-conditions). The most basic kinds of conditions are:

- **A hashlock condition:** One can fulfill a hashlock condition by providing the correct "preimage" (similar to a password or secret phrase)
- **A simple signature condition:** One can fulfill a simple signature condition by a providing a valid cryptographic signature (i.e. corresponding to the public key of an owner, usually)
- **A timeout condition:** Anyone can fulfill a timeout condition before the condition's expiry time. After the expiry time, nobody can fulfill the condition. Another way to say this is that a timeout condition's fulfillment is valid (TRUE) before the expiry time and invalid (FALSE) after the expiry time. Note: at the time of writing, timeout conditions are BigchainDB-specific (i.e. not part of the Interledger specs).

A more complex condition can be composed by using n of the above conditions as inputs to an m-of-n threshold condition (a logic gate which outputs TRUE iff m or more inputs are TRUE). If there are n inputs to a threshold condition:

- 1-of-n is the same as a logical OR of all the inputs
- n-of-n is the same as a logical AND of all the inputs

For example, one could create a condition requiring that m (of n) owners provide signatures before their asset can be transferred to new owners.

One can also put different weights on the inputs to threshold condition, along with a threshold that the weighted-sum-of-inputs must pass for the output to be TRUE. Weights could be used, for example, to express the number of shares that someone owns in an asset.

The (single) output of a threshold condition can be used as one of the inputs of other threshold conditions. This means that one can combine threshold conditions to build complex logical expressions, e.g. (x OR y) AND (u OR v).

Aside: In BigchainDB, the output of an m-of-n threshold condition can be inverted on the way out, so an output that would have been TRUE would get changed to FALSE (and vice versa). This enables the creation of NOT, NOR and NAND gates. At the time of writing, this "inverted threshold condition" is BigchainDB-specific (i.e. not part of the Interledger specs). It should only be used in combination with a timeout condition.

When one creates a condition, one can calculate its fulfillment length (e.g. 96). The more complex the condition, the larger its fulfillment length will be. A BigchainDB federation can put an upper limit on the allowed fulfillment length, as a way of capping the complexity of conditions (and the computing time required to validate them).

If someone tries to make a condition where the output of a threshold condition feeds into the input of another "earlier" threshold condition (i.e. in a closed logical circuit), then their computer will take forever to calculate the (infinite) "condition URI", at least in theory. In practice, their computer will run out of memory or their client software will timeout after a while.

Aside: In what follows, the list of `owners_after` (in a condition) is always who owned the asset at the time the transaction completed, but before the next transaction started. The list of `owners_before` (in a fulfillment) is always equal to the list of `owners_after` in that asset's previous transaction.

10.3.1 (Crypto-) Conditions

One New Owner

If there is only one *new owner*, the condition will be a simple signature condition (i.e. only one signature is required).

```
{
    "cid": "<condition index>",
    "condition": {
        "details": {
            "bitmask": "<base16 int>",
            "public_key": "<new owner public key>",
            "signature": null,
            "type": "fulfillment",
            "type_id": "<base16 int>"
        },
        "uri": "<string>"
    },
    "owners_after": ["<new owner public key>"],
    "amount": "<int>"
}
```

- **Condition header**:
  - `cid`: Condition index so that we can reference this output as an input to another transaction. It also matches the input `fid`, making this the condition to fulfill in order to spend the asset used as input with `fid`.
  - `owners_after`: A list containing one item: the public key of the new owner.
  - `amount`: The amount of shares for a divisible asset to send to the new owners.
- **Condition body**:
  - `bitmask`: A set of bits representing the features required by the condition type.
  - `public_key`: The new owner's public key.
  - `type_id`: The fulfillment type ID; see the ILP spec.
  - `uri`: Binary representation of the condition using only URL-safe characters.

Multiple New Owners

If there are multiple *new owners*, they can create a ThresholdCondition requiring a signature from each of them in order to spend the asset. For example:

```
{
    "cid": "<condition index>",
    "condition": {
        "details": {
            "bitmask": 41,
            "subfulfillments": [
                {
                    "bitmask": 32,
                    "public_key": "<new owner 1 public key>",
                    "signature": null,
                    "type": "fulfillment",
                    "type_id": 4,
```

```
                    "weight": 1
                },
                {
                    "bitmask": 32,
                    "public_key": "<new owner 2 public key>",
                    "signature": null,
                    "type": "fulfillment",
                    "type_id": 4,
                    "weight": 1
                }
            ],
            "threshold": 2,
            "type": "fulfillment",
            "type_id": 2
        },
        "uri": "cc:2:29:ytNK3X6-bZsbF-nCGDTuopUIMi1HCyCkyPewm6oLI3o:206"},
        "owners_after": [
            "owner 1 public key>",
            "owner 2 public key>"
        ]
}
```

- `subfulfillments`: a list of fulfillments
  - `weight`: integer weight for each subfulfillment's contribution to the threshold
- `threshold`: threshold to reach for the subfulfillments to reach a valid fulfillment

The `weights` and `threshold` could be adjusted. For example, if the `threshold` was changed to 1 above, then only one of the new owners would have to provide a signature to spend the asset.

10.3.2 Fulfillments

One Current Owner

If there is only one *current owner*, the fulfillment will be a simple signature fulfillment (i.e. containing just one signature).

```
{
    "owners_before": ["<public key of the owner before the transaction happened>"],
    "fid": 0,
    "fulfillment":
↪"cf:4:RxFzIE679tFBk8zwEgizhmTuciAylvTUwy6EL6ehddHFJOhK5F4IjwQ1xLu2oQK9iyRCZJdfWAefZVjTt3DeG5j2exqxp
↪UrCwgnj92dnFRAEE",
    "input": {
        "cid": 0,
        "txid": "11b3e7d893cc5fdfcf1a1706809c7def290a3b10b0bef6525d10b024649c42d3"
    }
}
```

- `fid`: Fulfillment index. It matches a `cid` in the conditions with a new *crypto-condition* that the new owner needs to fulfill to spend this asset.
- `owners_before`: A list of public keys of the owners before the transaction; in this case it has just one public key.
- `fulfillment`: A crypto-conditions URI that encodes the cryptographic fulfillments like signatures and others, see crypto-conditions.

- `input`: Pointer to the asset and condition of a previous transaction
  - `cid`: Condition index
  - `txid`: Transaction id

10.4 The Block Model

A block has the following structure:

```
{
  "id": "<hash of block>",
  "block": {
    "timestamp": "<block-creation timestamp>",
    "transactions": ["<list of transactions>"],
    "node_pubkey": "<public key of the node creating the block>",
    "voters": ["<list of federation nodes public keys>"]
  },
  "signature": "<signature of block>"
}
```

- `id`: The hash of the serialized `block` (i.e. the `timestamp`, `transactions`, `node_pubkey`, and `voters`). This is also a database primary key; that's how we ensure that all blocks are unique.
- **block:**
  - `timestamp`: The Unix time when the block was created. It's provided by the node that created the block.
  - `transactions`: A list of the transactions included in the block.
  - `node_pubkey`: The public key of the node that created the block.
  - `voters`: A list of the public keys of federation nodes at the time the block was created. It's the list of federation nodes which can cast a vote on this block. This list can change from block to block, as nodes join and leave the federation.
- `signature`: Cryptographic signature of the block by the node that created the block. (To create the signature, the node serializes the block contents and signs it with its private key.)

10.4.1 Working with Blocks

There's a **Block** class for creating and working with Block objects; look in /bigchaindb/models.py. (The link is to the latest version on the master branch on GitHub.)

10.5 The Vote Model

A vote has the following structure:

```
{
    "id": "<RethinkDB-generated ID for the vote>",
    "node_pubkey": "<the public key of the voting node>",
    "vote": {
        "voting_for_block": "<id of the block the node is voting for>",
        "previous_block": "<id of the block previous to this one>",
```

```
        "is_block_valid": "<true|false>",
        "invalid_reason": "<None|DOUBLE_SPEND|TRANSACTIONS_HASH_MISMATCH|NODES_
↪PUBKEYS_MISMATCH",
        "timestamp": "<Unix time when the vote was generated, provided by the voting␣
↪node>"
    },
    "signature": "<signature of vote>"
}
```


Note: The `invalid_reason` was not being used and may be dropped in a future version of BigchainDB. See Issue #217 on GitHub.

CHAPTER 11

Transaction Schema

- *Transaction*
- *Transaction Body*
- *Condition*
- *Fulfillment*
- *Asset*
- *Metadata*

11.1 Transaction

This is the outer transaction wrapper. It contains the ID, version and the body of the transaction, which is also called `transaction`.

11.1.1 Transaction.id

type: string

A sha3 digest of the transaction. The ID is calculated by removing all derived hashes and signatures from the transaction, serializing it to JSON with keys in sorted order and then hashing the resulting string with sha3.

11.1.2 Transaction.transaction

type: object

See: *Transaction Body*.

11.1.3 Transaction.version

type: integer

BigchainDB transaction schema version.

11.2 Transaction Body

See: *Transaction Body*.

11.2.1 Transaction.operation

type: string

Type of the transaction:

A `CREATE` transaction creates an asset in BigchainDB. This transaction has outputs (conditions) but no inputs (fulfillments), so a dummy fulfillment is used.

A `TRANSFER` transaction transfers ownership of an asset, by providing fulfillments to conditions of earlier transactions.

A `GENESIS` transaction is a special case transaction used as the sole member of the first block in a BigchainDB ledger.

11.2.2 Transaction.asset

type: object

Description of the asset being transacted.

See: *Asset*.

11.2.3 Transaction.fulfillments

type: array (object)

Array of the fulfillments (inputs) of a transaction.

See: *Fulfillment*.

11.2.4 Transaction.conditions

type: array (object)

Array of conditions (outputs) provided by this transaction.

See: *Condition*.

11.2.5 Transaction.metadata

type: object or null

User provided transaction metadata. This field may be `null` or may contain an id and an object with freeform metadata.

See: *Metadata*.

11.3 Condition

An output of a transaction. A condition describes a quantity of an asset and what conditions must be met in order for it to be fulfilled. See also: *fulfillment*.

11.3.1 Condition.cid

type: integer

Index of this condition's appearance in the *Transaction.conditions* array. In a transaction with 2 conditions, the ``cid``s will be 0 and 1.

11.3.2 Condition.condition

type: object

Body of the condition. Has the properties:

- **details**: Details of the condition.
- **uri**: Condition encoded as an ASCII string.

11.3.3 Condition.owners_after

type: array (string) or null

List of public keys associated with asset ownership at the time of the transaction.

11.3.4 Condition.amount

type: integer

Integral amount of the asset represented by this condition. In the case of a non divisible asset, this will always be 1.

11.4 Fulfillment

A fulfillment is an input to a transaction, named as such because it fulfills a condition of a previous transaction. In the case of a `CREATE` transaction, a fulfillment may provide no `input`.

11.4.1 Fulfillment.fid

type: integer

The offset of the fulfillment within the fulfillents array.

11.4.2 Fulfillment.owners_before

type: array (string) or null

List of public keys of the previous owners of the asset.

11.4.3 Fulfillment.fulfillment

type: object or string

Fulfillment of a *condition*, or put a different way, this is a payload that satisfies a condition in order to spend the associated asset.

11.4.4 Fulfillment.input

type: object or null

Reference to a condition of a previous transaction

11.5 Asset

Description of the asset being transacted. In the case of a `TRANSFER` transaction, this field contains only the ID of asset. In the case of a `CREATE` transaction, this field may contain properties:

11.5.1 Asset.id

type: string

A UUID of type 4 (random).

11.5.2 Asset.divisible

type: boolean

Whether or not the asset has a quantity that may be partially spent.

11.5.3 Asset.updatable

type: boolean

Whether or not the description of the asset may be updated. Defaults to false.

11.5.4 Asset.refillable

type: boolean

Whether the amount of the asset can change after its creation. Defaults to false.

11.5.5 Asset.data

type: object or null

User provided metadata associated with the asset. May also be `null`.

11.6 Metadata

User provided transaction metadata. This field may be `null` or may contain an id and an object with freeform metadata.

11.6.1 Metadata.id

type: string

A UUID of type 4 (random).

11.6.2 Metadata.data

type: object

User provided transaction metadata.

CHAPTER 12

Release Notes

You can find a list of all BigchainDB Server releases and release notes on GitHub at:

https://github.com/bigchaindb/bigchaindb/releases

The CHANGELOG.md file contains much the same information, but it also has notes about what to expect in the *next* release.

We also have a roadmap document in ROADMAP.md.

CHAPTER 13

Appendices

13.1 How to Install OS-Level Dependencies

BigchainDB Server has some OS-level dependencies that must be installed.

On Ubuntu 14.04 and 16.04, we found that the following was enough:

```
sudo apt-get update
sudo apt-get install g++ python3-dev libffi-dev
```

On Fedora 23 and 24, we found that the following was enough:

```
sudo dnf update
sudo dnf install gcc-c++ redhat-rpm-config python3-devel libffi-devel
```

(If you're using a version of Fedora before version 22, you may have to use `yum` instead of `dnf`.)

13.2 How to Install the Latest pip and setuptools

You can check the version of `pip` you're using (in your current virtualenv) by doing:

```
pip -V
```

If it says that `pip` isn't installed, or it says `pip` is associated with a Python version less than 3.4, then you must install a `pip` version associated with Python 3.4+. In the following instructions, we call it `pip3` but you may be able to use `pip` if that refers to the same thing. See the `pip` installation instructions.

On Ubuntu 14.04, we found that this works:

```
sudo apt-get install python3-pip
```

That should install a Python 3 version of `pip` named `pip3`. If that didn't work, then another way to get `pip3` is to do `sudo apt-get install python3-setuptools` followed by `sudo easy_install3 pip`.

You can upgrade `pip` (`pip3`) and `setuptools` to the latest versions using:

```
pip3 install --upgrade pip setuptools
```

13.3 Run BigchainDB with Docker

NOT for Production Use

For those who like using Docker and wish to experiment with BigchainDB in non-production environments, we currently maintain a Docker image and a `Dockerfile` that can be used to build an image for `bigchaindb`.

13.3.1 Pull and Run the Image from Docker Hub

Assuming you have Docker installed, you would proceed as follows.

In a terminal shell, pull the latest version of the BigchainDB Docker image using:

```
docker pull bigchaindb/bigchaindb
```

then do a one-time configuration step to create the config file; we will use the `-y` option to accept all the default values. The configuration file will be stored in a file on your host machine at `~/bigchaindb_docker/.bigchaindb`:

```
docker run --rm -v "$HOME/bigchaindb_docker:/data" -ti \
  bigchaindb/bigchaindb -y configure
Generating keypair
Configuration written to /data/.bigchaindb
Ready to go!
```

Let's analyze that command:

- `docker run` tells Docker to run some image
- `--rm` remove the container once we are done
- `-v "$HOME/bigchaindb_docker:/data"` map the host directory `$HOME/bigchaindb_docker` to the container directory `/data`; this allows us to have the data persisted on the host machine, you can read more in the official Docker documentation
- `-t` allocate a pseudo-TTY
- `-i` keep STDIN open even if not attached
- `bigchaindb/bigchaindb` the image to use
- `-y configure` execute the `configure` sub-command (of the `bigchaindb` command) inside the container, with the `-y` option to automatically use all the default config values

After configuring the system, you can run BigchainDB with the following command:

```
docker run -v "$HOME/bigchaindb_docker:/data" -d \
  --name bigchaindb \
  -p "58080:8080" -p "59984:9984" \
  bigchaindb/bigchaindb start
```

The command is slightly different from the previous one, the differences are:

- `-d` run the container in the background
- `--name bigchaindb` give a nice name to the container so it's easier to refer to it later

- `-p "58080:8080"` map the host port `58080` to the container port `8080` (the RethinkDB admin interface)
- `-p "59984:9984"` map the host port `59984` to the container port `9984` (the BigchainDB API server)
- `start` start the BigchainDB service

Another way to publish the ports exposed by the container is to use the `-P` (or `--publish-all`) option. This will publish all exposed ports to random ports. You can always run `docker ps` to check the random mapping.

You can also access the RethinkDB dashboard at http://localhost:58080/

If that doesn't work, then replace `localhost` with the IP or hostname of the machine running the Docker engine. If you are running docker-machine (e.g. on Mac OS X) this will be the IP of the Docker machine (`docker-machine ip machine_name`).

Load Testing with Docker

Now that we have BigchainDB running in the Docker container named `bigchaindb`, we can start another BigchainDB container to generate a load test for it.

First, make sure the container named `bigchaindb` is still running. You can check that using:

```
docker ps
```

You should see a container named `bigchaindb` in the list.

You can load test the BigchainDB running in that container by running the `bigchaindb load` command in a second container:

```
docker run --rm -v "$HOME/bigchaindb_docker:/data" \
  -e BIGCHAINDB_DATABASE_HOST=bigchaindb \
  --link bigchaindb \
  bigchaindb/bigchaindb load
```

Note the `--link` option to link to the first container (named `bigchaindb`).

Aside: The `bigchaindb load` command has several options (e.g. `-m`). You can read more about it in the documentation about the BigchainDB command line interface.

If you look at the RethinkDB dashboard (in your web browser), you should see the effects of the load test. You can also see some effects in the Docker logs using:

```
docker logs -f bigchaindb
```

13.3.2 Building Your Own Image

Assuming you have Docker installed, you would proceed as follows.

In a terminal shell:

```
git clone git@github.com:bigchaindb/bigchaindb.git
```

Build the Docker image:

```
docker build --tag local-bigchaindb .
```

Now you can use your own image to run BigchainDB containers.

13.4 JSON Serialization

We needed to clearly define how to serialize a JSON object to calculate the hash.

The serialization should produce the same byte output independently of the architecture running the software. If there are differences in the serialization, hash validations will fail although the transaction is correct.

For example, consider the following two methods of serializing `{'a': 1}`:

```
# Use a serializer provided by RethinkDB
a = r.expr({'a': 1}).to_json().run(b.connection)
u'{"a":1}'

# Use the serializer in Python's json module
b = json.dumps({'a': 1})
'{"a": 1}'

a == b
False
```

The results are not the same. We want a serialization and deserialization so that the following is always true:

```
deserialize(serialize(data)) == data
True
```

Since BigchainDB performs a lot of serialization we decided to use python-rapidjson which is a python wrapper for rapidjson a fast and fully RFC complient JSON parser.

```
import rapidjson

rapidjson.dumps(data, skipkeys=False,
                ensure_ascii=False,
                sort_keys=True)
```

- `skipkeys`: With skipkeys `False` if the provided keys are not a string the serialization will fail. This way we enforce all keys to be strings
- `ensure_ascii`: The RFC recommends `utf-8` for maximum interoperability. By setting `ensure_ascii` to `False` we allow unicode characters and python-rapidjson forces the encoding to `utf-8`.
- `sort_keys`: Sorted output by keys.

Every time we need to perform some operation on the data like calculating the hash or signing/verifying the transaction, we need to use the previous criteria to serialize the data and then use the `byte` representation of the serialized data (if we treat the data as bytes we eliminate possible encoding errors e.g. unicode characters). For example:

```
# calculate the hash of a transaction
# the transaction is a dictionary
tx_serialized = bytes(serialize(tx))
tx_hash = hashlib.sha3_256(tx_serialized).hexdigest()

# signing a transaction
tx_serialized = bytes(serialize(tx))
signature = sk.sign(tx_serialized)

# verify signature
tx_serialized = bytes(serialize(tx))
pk.verify(signature, tx_serialized)
```

13.5 Cryptography

The section documents the cryptographic algorithms and Python implementations that we use.

Before hashing or computing the signature of a JSON document, we serialize it as described in the section on JSON serialization.

13.5.1 Hashes

We compute hashes using the SHA3-256 algorithm and pysha3 as the Python implementation. We store the hex-encoded hash in the database. For example:

```
import hashlib
# monkey patch hashlib with sha3 functions
import sha3

data = "message"
tx_hash = hashlib.sha3_256(data).hexdigest()
```

13.5.2 Signature Algorithm and Keys

BigchainDB uses the Ed25519 public-key signature system for generating its public/private key pairs. Ed25519 is an instance of the Edwards-curve Digital Signature Algorithm (EdDSA). As of April 2016, EdDSA was in "Internet-Draft" status with the IETF but was already widely used.

BigchainDB uses the the ed25519 Python package, overloaded by the cryptoconditions library.

All keys are represented with the base58 encoding by default.

13.6 The Bigchain class

The Bigchain class is the top-level Python API for BigchainDB. If you want to create and initialize a BigchainDB database, you create a Bigchain instance (object). Then you can use its various methods to create transactions, write transactions (to the object/database), read transactions, etc.

class `bigchaindb.`**`Bigchain`**(*host=None*, *port=None*, *dbname=None*, *backend=None*, *public_key=None*, *private_key=None*, *keyring=[]*, *backlog_reassign_delay=None*)

Bigchain API

Create, read, sign, write transactions to the database

`__init__`(*host=None*, *port=None*, *dbname=None*, *backend=None*, *public_key=None*, *private_key=None*, *keyring=[]*, *backlog_reassign_delay=None*)

Initialize the Bigchain instance

A Bigchain instance has several configuration parameters (e.g. host). If a parameter value is passed as an argument to the Bigchain __init__ method, then that is the value it will have. Otherwise, the parameter value will come from an environment variable. If that environment variable isn't set, then the value will come from the local configuration file. And if that variable isn't in the local configuration file, then the parameter will have its default value (defined in bigchaindb.__init__).

Parameters

- **`host`** (*`str`*) – hostname where RethinkDB is running.

- **port** (*int*) – port in which RethinkDB is running (usually 28015).
- **dbname** (*str*) – the name of the database to connect to (usually bigchain).
- **backend** (`RehinkDBBackend`) – the database backend to use.
- **public_key** (*str*) – the base58 encoded public key for the ED25519 curve.
- **private_key** (*str*) – the base58 encoded private key for the ED25519 curve.
- **keyring** (*list[str]*) – list of base58 encoded public keys of the federation nodes.

write_transaction(*signed_transaction*, *durability='soft'*)

Write the transaction to bigchain.

When first writing a transaction to the bigchain the transaction will be kept in a backlog until it has been validated by the nodes of the federation.

Parameters **signed_transaction** (*Transaction*) – transaction with the *signature* included.

Returns database response

Return type dict

reassign_transaction(*transaction*)

Assign a transaction to a new node

Parameters **transaction** (*dict*) – assigned transaction

Returns database response or None if no reassignment is possible

Return type dict

delete_transaction(**transaction_id*)

Delete a transaction from the backlog.

Parameters ***transaction_id** (*str*) – the transaction(s) to delete

Returns The database response.

get_stale_transactions()

Get a cursor of stale transactions.

Transactions are considered stale if they have been assigned a node, but are still in the backlog after some amount of time specified in the configuration

validate_transaction(*transaction*)

Validate a transaction.

Parameters **transaction** (*Transaction*) – transaction to validate.

Returns The transaction if the transaction is valid else it raises an exception describing the reason why the transaction is invalid.

is_valid_transaction(*transaction*)

Check whether a transaction is valid or invalid.

Similar to `validate_transaction()` but never raises an exception. It returns `False` if the transaction is invalid.

Parameters **transaction** (`Transaction`) – transaction to check.

Returns The `Transaction` instance if valid, otherwise `False`.

get_block(*block_id*, *include_status=False*)
Get the block with the specified *block_id* (and optionally its status)

Returns the block corresponding to *block_id* or None if no match is found.

Parameters

- **block_id** (*str*) – transaction id of the transaction to get
- **include_status** (*bool*) – also return the status of the block the return value is then a tuple: (block, status)

get_transaction(*txid*, *include_status=False*)
Get the transaction with the specified *txid* (and optionally its status)

This query begins by looking in the bigchain table for all blocks containing a transaction with the specified *txid*. If one of those blocks is valid, it returns the matching transaction from that block. Else if some of those blocks are undecided, it returns a matching transaction from one of them. If the transaction was found in invalid blocks only, or in no blocks, then this query looks for a matching transaction in the backlog table, and if it finds one there, it returns that.

Parameters

- **txid** (*str*) – transaction id of the transaction to get
- **include_status** (*bool*) – also return the status of the transaction the return value is then a tuple: (tx, status)

Returns A `Transaction` instance if the transaction was found in a valid block, an undecided block, or the backlog table, otherwise `None`. If `include_status` is `True`, also returns the transaction's status if the transaction was found.

get_status(*txid*)
Retrieve the status of a transaction with *txid* from bigchain.

Parameters **txid** (*str*) – transaction id of the transaction to query

Returns transaction status ('valid', 'undecided', or 'backlog'). If no transaction with that *txid* was found it returns *None*

Return type (string)

get_blocks_status_containing_tx(*txid*)
Retrieve block ids and statuses related to a transaction

Transactions may occur in multiple blocks, but no more than one valid block.

Parameters **txid** (*str*) – transaction id of the transaction to query

Returns A dict of blocks containing the transaction, e.g. {block_id_1: 'valid', block_id_2: 'invalid' ...}, or None

get_transaction_by_metadata_id(*metadata_id*)
Retrieves valid or undecided transactions related to a particular metadata.

When creating a transaction one of the optional arguments is the *metadata*. The metadata is a generic dict that contains extra information that can be appended to the transaction.

To make it easy to query the bigchain for that particular metadata we create a UUID for the metadata and store it with the transaction.

Parameters **metadata_id** (*str*) – the id for this particular metadata.

Returns A list of valid or undecided transactions containing that metadata. If no transaction exists with that metadata it returns an empty list *[]*

get_transactions_by_asset_id(*asset_id*)
Retrieves valid or undecided transactions related to a particular asset.

A digital asset in bigchaindb is identified by an uuid. This allows us to query all the transactions related to a particular digital asset, knowing the id.

Parameters **asset_id** (*str*) – the id for this particular asset.

Returns A list of valid or undecided transactions related to the asset. If no transaction exists for that asset it returns an empty list *[]*

get_asset_by_id(*asset_id*)
Returns the asset associated with an asset_id.

Parameters **asset_id** (*str*) – The asset id.

Returns `Asset` if the asset exists else None.

get_spent(*txid*, *cid*)
Check if a *txid* was already used as an input.

A transaction can be used as an input for another transaction. Bigchain needs to make sure that a given *txid* is only used once.

Parameters

- **txid** (*str*) – The id of the transaction
- **cid** (*num*) – the index of the condition in the respective transaction

Returns The transaction (Transaction) that used the *txid* as an input else *None*

get_owned_ids(*owner*)
Retrieve a list of 'txid's that can be used as inputs.

Parameters **owner** (*str*) – base58 encoded public key.

Returns list of 'txid's and 'cid's pointing to another transaction's condition

Return type `list` of TransactionLink

create_block(*validated_transactions*)
Creates a block given a list of *validated_transactions*.

Note that this method does not validate the transactions. Transactions should be validated before calling create_block.

Parameters **validated_transactions** (*list(Transaction)*) – list of validated transactions.

Returns created block.

Return type Block

validate_block(*block*)
Validate a block.

Parameters **block** (*Block*) – block to validate.

Returns The block if the block is valid else it raises and exception describing the reason why the block is invalid.

has_previous_vote(*block_id*, *voters*)
Check for previous votes from this node

Parameters

- **block_id** (*str*) – the id of the block to check
- **voters** (*list (str)*) – the voters of the block to check

Returns `True` if this block already has a valid vote from this node, `False` otherwise.

Return type bool

Raises `ImproperVoteError` – If there is already a vote, but the vote is invalid.

write_block (*block*, *durability='soft'*)
Write a block to bigchain.

Parameters **block** (*Block*) – block to write to bigchain.

prepare_genesis_block ()
Prepare a genesis block.

create_genesis_block ()
Create the genesis block

Block created when bigchain is first initialized. This method is not atomic, there might be concurrency problems if multiple instances try to write the genesis block when the BigchainDB Federation is started, but it's a highly unlikely scenario.

vote (*block_id*, *previous_block_id*, *decision*, *invalid_reason=None*)
Create a signed vote for a block given the `previous_block_id` and the `decision` (valid/invalid).

Parameters

- **block_id** (*str*) – The id of the block to vote on.
- **previous_block_id** (*str*) – The id of the previous block.
- **decision** (*bool*) – Whether the block is valid or invalid.
- **invalid_reason** (*Optional[str]*) – Reason the block is invalid

write_vote (*vote*)
Write the vote to the database.

get_last_voted_block ()
Returns the last block that this node voted on.

get_unvoted_blocks ()
Return all the blocks that have not been voted on by this node.

Returns a list of unvoted blocks

Return type `list` of `dict`

block_election_status (*block_id*, *voters*)
Tally the votes on a block, and return the status: valid, invalid, or undecided.

13.7 Consensus

class `bigchaindb.consensus.`**BaseConsensusRules**
Base consensus rules for Bigchain.

static validate_transaction (*bigchain*, *transaction*)
See `bigchaindb.models.Transaction.validate()` for documentation.

static validate_block (*bigchain*, *block*)
See `bigchaindb.models.Block.validate()` for documentation.

static verify_vote_signature (*voters*, *signed_vote*)
Verify the signature of a vote.

Refer to the documentation of `bigchaindb.util.verify_signature()`.

13.8 Pipelines

13.8.1 Block Creation

This module takes care of all the logic related to block creation.

The logic is encapsulated in the `BlockPipeline` class, while the sequence of actions to do on transactions is specified in the `create_pipeline` function.

class `bigchaindb.pipelines.block.`**BlockPipeline**
This class encapsulates the logic to create blocks.

Note: Methods of this class will be executed in different processes.

---

filter_tx (*tx*)
Filter a transaction.

Parameters **tx** (*dict*) – the transaction to process.

Returns The transaction if assigned to the current node, `None` otherwise.

Return type dict

validate_tx (*tx*)
Validate a transaction.

Also checks if the transaction already exists in the blockchain. If it does, or it's invalid, it's deleted from the backlog immediately.

Parameters **tx** (*dict*) – the transaction to validate.

Returns The transaction if valid, `None` otherwise.

Return type `Transaction`

create (*tx*, *timeout=False*)
Create a block.

This method accumulates transactions to put in a block and outputs a block when one of the following conditions is true: - the size limit of the block has been reached, or - a timeout happened.

Parameters

- **tx** (`Transaction`) – the transaction to validate, might be None if a timeout happens.
- **timeout** (*bool*) – `True` if a timeout happened (Default: `False`).

Returns The block, if a block is ready, or `None`.

Return type `Block`

write (*block*)
Write the block to the Database.

Parameters **block** (`Block`) – the block of transactions to write to the database.

Returns The Block.

Return type `Block`

`delete_tx`(*block*)
Delete transactions.

Parameters **`block`** (`Block`) – the block containg the transactions to delete.

Returns The block.

Return type `Block`

`bigchaindb.pipelines.block.`**`initial`**()
Return old transactions from the backlog.

`bigchaindb.pipelines.block.`**`get_changefeed`**()
Create and return the changefeed for the backlog.

`bigchaindb.pipelines.block.`**`create_pipeline`**()
Create and return the pipeline of operations to be distributed on different processes.

`bigchaindb.pipelines.block.`**`start`**()
Create, start, and return the block pipeline.

13.8.2 Block Voting

This module takes care of all the logic related to block voting.

The logic is encapsulated in the `Vote` class, while the sequence of actions to do on transactions is specified in the `create_pipeline` function.

class `bigchaindb.pipelines.vote.`**`Vote`**
This class encapsulates the logic to vote on blocks.

Note: Methods of this class will be executed in different processes.

`ungroup`(*block_id*, *transactions*)
Given a block, ungroup the transactions in it.

Parameters

- **`block_id`** (*`str`*) – the id of the block in progress.
- **`transactions`** (*`list(Transaction)`*) – transactions of the block in progress.

Returns `None` if the block has been already voted, an iterator that yields a transaction, block id, and the total number of transactions contained in the block otherwise.

`validate_tx`(*tx*, *block_id*, *num_tx*)
Validate a transaction.

Parameters

- **`tx`** (*`dict`*) – the transaction to validate
- **`block_id`** (*`str`*) – the id of block containing the transaction
- **`num_tx`** (*`int`*) – the total number of transactions to process

Returns Three values are returned, the validity of the transaction, `block_id`, `num_tx`.

vote (*tx_validity*, *block_id*, *num_tx*)
Collect the validity of transactions and cast a vote when ready.

Parameters

- **tx_validity** (*bool*) – the validity of the transaction
- **block_id** (*str*) – the id of block containing the transaction
- **num_tx** (*int*) – the total number of transactions to process

Returns None, or a vote if a decision has been reached.

write_vote (*vote*)
Write vote to the database.

Parameters **vote** – the vote to write.

`bigchaindb.pipelines.vote.`**initial**()
Return unvoted blocks.

`bigchaindb.pipelines.vote.`**get_changefeed**()
Create and return the changefeed for the bigchain table.

`bigchaindb.pipelines.vote.`**create_pipeline**()
Create and return the pipeline of operations to be distributed on different processes.

`bigchaindb.pipelines.vote.`**start**()
Create, start, and return the block pipeline.

13.8.3 Block Status

This module takes care of all the logic related to block status.

Specifically, what happens when a block becomes invalid. The logic is encapsulated in the `Election` class, while the sequence of actions is specified in `create_pipeline`.

class `bigchaindb.pipelines.election.`**Election**
Election class.

check_for_quorum (*next_vote*)
Checks if block has enough invalid votes to make a decision

Parameters **next_vote** – The next vote.

requeue_transactions (*invalid_block*)
Liquidates transactions from invalid blocks so they can be processed again

13.8.4 Stale Transaction Monitoring

This module monitors for stale transactions.

It reassigns transactions which have been assigned a node but remain in the backlog past a certain amount of time.

class `bigchaindb.pipelines.stale.`**StaleTransactionMonitor** (*timeout=5*, *backlog_reassign_delay=None*)
This class encapsulates the logic for re-assigning stale transactions.

Note: Methods of this class will be executed in different processes.

check_transactions()
Poll backlog for stale transactions

Returns txs to be re assigned

Return type txs (list)

reassign_transactions(*tx*)
Put tx back in backlog with new assignee

Returns transaction

bigchaindb.pipelines.stale.**create_pipeline**(*timeout=5*, *backlog_reassign_delay=5*)
Create and return the pipeline of operations to be distributed on different processes.

bigchaindb.pipelines.stale.**start**(*timeout=5*, *backlog_reassign_delay=None*)
Create, start, and return the block pipeline.

13.8.5 Utilities

Utility classes and functions to work with the pipelines.

class bigchaindb.pipelines.utils.**ChangeFeed**(*table*, *operation*, *prefeed=None*, *bigchain=None*)
This class wraps a RethinkDB changefeed adding a `prefeed`.

It extends `multipipes.Node` to make it pluggable in other Pipelines instances, and makes usage of `self.outqueue` to output the data.

A changefeed is a real time feed on inserts, updates, and deletes, and is volatile. This class is a helper to create changefeeds. Moreover, it provides a way to specify a `prefeed` of iterable data to output before the actual changefeed.

13.9 Basic AWS Setup

Before you can deploy anything on AWS, you must do a few things.

13.9.1 Get an AWS Account

If you don't already have an AWS account, you can sign up for one for free at aws.amazon.com.

13.9.2 Install the AWS Command-Line Interface

To install the AWS Command-Line Interface (CLI), just do:

```
pip install awscli
```

13.9.3 Create an AWS Access Key

The next thing you'll need is an AWS access key. If you don't have one, you can create one using the instructions in the AWS documentation. You should get an access key ID (e.g. AKIAIOSFODNN7EXAMPLE) and a secret access key (e.g. wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY).

You should also pick a default AWS region name (e.g. `eu-central-1`). That's where your cluster will run. The AWS documentation has a list of them.

Once you've got your AWS access key, and you've picked a default AWS region name, go to a terminal session and enter:

```
aws configure
```

and answer the four questions. For example:

```
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: eu-central-1
Default output format [None]: [Press Enter]
```

This writes two files: `~/.aws/credentials` and `~/.aws/config`. AWS tools and packages look for those files.

13.9.4 Generate an RSA Key Pair for SSH

Eventually, you'll have one or more instances (virtual machines) running on AWS and you'll want to SSH to them. To do that, you need a public/private key pair. The public key will be sent to AWS, and you can tell AWS to put it in any instances you provision there. You'll keep the private key on your local workstation.

First you need to make up a key name. Some ideas:

- `bcdb-troy-1`
- `bigchaindb-7`
- `bcdb-jupiter`

If you already have key pairs on AWS (Amazon EC2), you have to pick a name that's not already being used. Below, replace every instance of `<key-name>` with your actual key name. To generate a public/private RSA key pair with that name:

```
ssh-keygen -t rsa -C "<key-name>" -f ~/.ssh/<key-name>
```

It will ask you for a passphrase. You can use whatever passphrase you like, but don't lose it. Two keys (files) will be created in `~/.ssh/`:

1. `~/.ssh/<key-name>.pub` is the public key
2. `~/.ssh/<key-name>` is the private key

To send the public key to AWS, use the AWS Command-Line Interface:

```
aws ec2 import-key-pair \
--key-name "<key-name>" \
--public-key-material file://~/.ssh/<key-name>.pub
```

If you're curious why there's a `file://` in front of the path to the public key, see issue aws/aws-cli#41 on GitHub.

If you want to verify that your key pair was imported by AWS, go to the Amazon EC2 console at https://console.aws.amazon.com/ec2/, select the region you gave above when you did `aws configure` (e.g. eu-central-1), click on **Key Pairs** in the left sidebar, and check that `<key-name>` is listed.

13.10 Notes for Firewall Setup

This is a page of notes on the ports potentially used by BigchainDB nodes and the traffic they should expect, to help with firewall setup (and security group setup on AWS). This page is *not* a firewall tutorial or step-by-step guide.

13.10.1 Expected Unsolicited Inbound Traffic

Assuming you aren't exposing the RethinkDB web interface on port 8080 (or any other port, because there are more secure ways to access it), there are only three ports that should expect unsolicited inbound traffic:

1. **Port 22** can expect inbound SSH (TCP) traffic from the node administrator (i.e. a small set of IP addresses).
2. **Port 9984** can expect inbound HTTP (TCP) traffic from BigchainDB clients sending transactions to the BigchainDB HTTP API.
3. **Port 29015** can expect inbound TCP traffic from other RethinkDB nodes in the RethinkDB cluster (for RethinkDB intracluster communications).

All other ports should only get inbound traffic in response to specific requests from inside the node.

13.10.2 Port 22

Port 22 is the default SSH port (TCP) so you'll at least want to make it possible to SSH in from your remote machine(s).

13.10.3 Port 53

Port 53 is the default DNS port (UDP). It may be used, for example, by some package managers when look up the IP address associated with certain package sources.

13.10.4 Port 80

Port 80 is the default HTTP port (TCP). It's used by some package managers to get packages. It's *not* used by the RethinkDB web interface (see Port 8080 below) or the BigchainDB client-server HTTP API (Port 9984).

13.10.5 Port 123

Port 123 is the default NTP port (UDP). You should be running an NTP daemon on production BigchainDB nodes. NTP daemons must be able to send requests to external NTP servers and accept the respones.

13.10.6 Port 161

Port 161 is the default SNMP port (usually UDP, sometimes TCP). SNMP is used, for example, by some server monitoring systems.

13.10.7 Port 443

Port 443 is the default HTTPS port (TCP). You may need to open it up for outbound requests (and inbound responses) temporarily because some RethinkDB installation instructions use wget over HTTPS to get the RethinkDB GPG key. Package managers might also get some packages using HTTPS.

13.10.8 Port 8125

If you set up a cluster-monitoring server, then StatsD will send UDP packets to Telegraf (on the monitoring server) via port 8125.

13.10.9 Port 8080

Port 8080 is the default port used by RethinkDB for its adminstrative web (HTTP) interface (TCP). While you *can*, you shouldn't allow traffic arbitrary external sources. You can still use the RethinkDB web interface by binding it to localhost and then accessing it via a SOCKS proxy or reverse proxy; see "Binding the web interface port" on the RethinkDB page about securing your cluster.

13.10.10 Port 9984

Port 9984 is the default port for the BigchainDB client-server HTTP API (TCP), which is served by Gunicorn HTTP Server. It's *possible* allow port 9984 to accept inbound traffic from anyone, but we recommend against doing that. Instead, set up a reverse proxy server (e.g. using Nginx) and only allow traffic from there. Information about how to do that can be found in the Gunicorn documentation. (They call it a proxy.)

If Gunicorn and the reverse proxy are running on the same server, then you'll have to tell Gunicorn to listen on some port other than 9984 (so that the reverse proxy can listen on port 9984). You can do that by setting `server.bind` to 'localhost:PORT' in the BigchainDB Configuration Settings, where PORT is whatever port you chose (e.g. 9983).

You may want to have Gunicorn and the reverse proxy running on different servers, so that both can listen on port 9984. That would also help isolate the effects of a denial-of-service attack.

13.10.11 Port 28015

Port 28015 is the default port used by RethinkDB client driver connections (TCP). If your BigchainDB node is just one server, then Port 28015 only needs to listen on localhost, because all the client drivers will be running on localhost. Port 28015 doesn't need to accept inbound traffic from the outside world.

13.10.12 Port 29015

Port 29015 is the default port for RethinkDB intracluster connections (TCP). It should only accept incoming traffic from other RethinkDB servers in the cluster (a list of IP addresses that you should be able to find out).

13.10.13 Other Ports

On Linux, you can use commands such as `netstat -tunlp` or `lsof -i` to get a sense of currently open/listening ports and connections, and the associated processes.

13.10.14 Cluster-Monitoring Server

If you set up a cluster-monitoring server (running Telegraf, InfluxDB & Grafana), Telegraf will listen on port 8125 for UDP packets from StatsD, and the Grafana web dashboard will use port 3000. (Those are the default ports.)

13.11 Notes on NTP Daemon Setup

There are several NTP daemons available, including:

- The reference NTP daemon (`ntpd`) from ntp.org; see their support website
- chrony
- OpenNTPD
- Maybe NTPsec, once it's production-ready
- Maybe Ntimed, once it's production-ready
- More

We suggest you run your NTP daemon in a mode which will tell your OS kernel to handle leap seconds in a particular way: the default NTP way, so that system clock adjustments are localized and not spread out across the minutes, hours, or days surrounding leap seconds (e.g. "slewing" or "smearing"). There's a nice Red Hat Developer Blog post about the various options.

Use the default mode with `ntpd` and `chronyd`. For another NTP daemon, consult its documentation.

It's tricky to make an NTP daemon setup secure. Always install the latest version and read the documentation about how to configure and run it securely. See the notes on firewall setup.

13.11.1 Amazon Linux Instances

If your BigchainDB node is running on an Amazon Linux instance (i.e. a Linux instance packaged by Amazon, not Canonical, Red Hat, or someone else), then an NTP daemon should already be installed and configured. See the EC2 documentation on Setting the Time for Your Linux Instance.

That said, you should check *which* NTP daemon is installed. Is it recent? Is it configured securely?

13.11.2 Ubuntu's ntp Package

The Ubuntu 14.04 (Trusty Tahr) package `ntp` is based on the reference implementation of an NTP daemon (i.e. `ntpd`).

The following commands will uninstall the `ntp` and `ntpdate` packages, install the latest `ntp` package (which *might not be based on the latest ntpd code*), and start the NTP daemon (a local NTP server). (`ntpdate` is not reinstalled because it's deprecated and you shouldn't use it.)

```
sudo apt-get --purge remove ntp ntpdate
sudo apt-get autoremove
sudo apt-get update
sudo apt-get install ntp
# That should start the NTP daemon too, but just to be sure:
sudo service ntp restart
```

You can check if `ntpd` is running using `sudo ntpq -p`.

You may want to use different NTP time servers. You can change them by editing the NTP config file `/etc/ntp.conf`.

Note: A server running an NTP daemon can be used by others for DRDoS amplification attacks. The above installation procedure should install a default NTP configuration file `/etc/ntp.conf` with the lines:

```
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
```

Those lines should prevent the NTP daemon from being used in an attack. (The first line is for IPv4, the second for IPv6.)

There are additional things you can do to make NTP more secure. See the NTP Support Website for more details.

13.12 Example RethinkDB Storage Setups

13.12.1 Example Amazon EC2 Setups

We have some scripts for deploying a *test* BigchainDB cluster on AWS. Those scripts include command sequences to set up storage for RethinkDB. In particular, look in the file /deploy-cluster-aws/fabfile.py, under `def prep_rethinkdb_storage(USING_EBS)`. Note that there are two cases:

1. **Using EBS (Amazon Elastic Block Store).** This is always an option, and for some instance types ("EBS-only"), it's the only option.
2. **Using an "instance store" volume provided with an Amazon EC2 instance.** Note that our scripts only use one of the (possibly many) volumes in the instance store.

There's some explanation of the steps in the Amazon EC2 documentation about making an Amazon EBS volume available for use.

You shouldn't use an EC2 "instance store" to store RethinkDB data for a production node, because it's not replicated and it's only intended for temporary, ephemeral data. If the associated instance crashes, is stopped, or is terminated, the data in the instance store is lost forever. Amazon EBS storage is replicated, has incremental snapshots, and is low-latency.

13.12.2 Example Using Amazon EFS

TODO

13.12.3 Other Examples?

TODO

Maybe RAID, ZFS, ... (over EBS volumes, i.e. a DIY Amazon EFS)

13.13 Licenses

Information about how the BigchainDB Server code and documentation are licensed can be found in the LICENSES.md file of the bigchaindb/bigchaindb repository on GitHub.

13.14 Installing BigchainDB on LXC containers using LXD

You can visit this link to install LXD (instructions here): LXD Install

(assumption is that you are using Ubuntu 14.04 for host/container)

Let us create an LXC container (via LXD) with the following command:

`lxc launch ubuntu:14.04 bigchaindb`

(ubuntu:14.04 - this is the remote server the command fetches the image from) (bigchaindb - is the name of the container)

Below is the `install.sh` script you will need to install BigchainDB within your container.

Here is my `install.sh`:

```
#!/bin/bash
set -ex
export DEBIAN_FRONTEND=noninteractive
apt-get install -y wget
source /etc/lsb-release && echo "deb http://download.rethinkdb.com/apt $DISTRIB_
↪CODENAME main" | sudo tee /etc/apt/sources.list.d/rethinkdb.list
wget -qO- https://download.rethinkdb.com/apt/pubkey.gpg | sudo apt-key add -
apt-get update
apt-get install -y rethinkdb python3-pip
pip3 install --upgrade pip wheel setuptools
pip install ptpython bigchaindb
```

Copy/Paste the above `install.sh` into the directory/path you are going to execute your LXD commands from (ie. the host).

Make sure your container is running by typing:

`lxc list`

Now, from the host (and the correct directory) where you saved `install.sh`, run this command:

`cat install.sh | lxc exec bigchaindb /bin/bash`

If you followed the commands correctly, you will have successfully created an LXC container (using LXD) that can get you up and running with BigchainDB in <5 minutes (depending on how long it takes to download all the packages).

Python Module Index

b

bigchaindb.consensus, 75
bigchaindb.pipelines.block, 76
bigchaindb.pipelines.election, 78
bigchaindb.pipelines.stale, 78
bigchaindb.pipelines.utils, 79
bigchaindb.pipelines.vote, 77

/transactions

GET /transactions/{tx_id}, 32
GET /transactions/{tx_id}/status, 32
POST /transactions/, 30

/unspents?owner_after={owner_after}

GET /unspents?owner_after={owner_after}, 34

Index

Symbols

__init__() (bigchaindb.core.Bigchain method), 71

B

BaseConsensusRules (class in bigchaindb.consensus), 75
Bigchain (class in bigchaindb), 71
bigchaindb.consensus (module), 75
bigchaindb.pipelines.block (module), 76
bigchaindb.pipelines.election (module), 78
bigchaindb.pipelines.stale (module), 78
bigchaindb.pipelines.utils (module), 79
bigchaindb.pipelines.vote (module), 77
block_election_status() (bigchaindb.Bigchain method), 75
BlockPipeline (class in bigchaindb.pipelines.block), 76

C

ChangeFeed (class in bigchaindb.pipelines.utils), 79
check_for_quorum() (bigchaindb.pipelines.election.Election method), 78
check_transactions() (bigchaindb.pipelines.stale.StaleTransactionMonitor method), 78
create() (bigchaindb.pipelines.block.BlockPipeline method), 76
create_block() (bigchaindb.Bigchain method), 74
create_genesis_block() (bigchaindb.Bigchain method), 75
create_pipeline() (in module bigchaindb.pipelines.block), 77
create_pipeline() (in module bigchaindb.pipelines.stale), 79
create_pipeline() (in module bigchaindb.pipelines.vote), 78

D

delete_transaction() (bigchaindb.Bigchain method), 72
delete_tx() (bigchaindb.pipelines.block.BlockPipeline method), 77

E

Election (class in bigchaindb.pipelines.election), 78

F

filter_tx() (bigchaindb.pipelines.block.BlockPipeline method), 76

G

get_asset_by_id() (bigchaindb.Bigchain method), 74
get_block() (bigchaindb.Bigchain method), 72
get_blocks_status_containing_tx() (bigchaindb.Bigchain method), 73
get_changefeed() (in module bigchaindb.pipelines.block), 77
get_changefeed() (in module bigchaindb.pipelines.vote), 78
get_last_voted_block() (bigchaindb.Bigchain method), 75
get_owned_ids() (bigchaindb.Bigchain method), 74
get_spent() (bigchaindb.Bigchain method), 74
get_stale_transactions() (bigchaindb.Bigchain method), 72
get_status() (bigchaindb.Bigchain method), 73
get_transaction() (bigchaindb.Bigchain method), 73
get_transaction_by_metadata_id() (bigchaindb.Bigchain method), 73
get_transactions_by_asset_id() (bigchaindb.Bigchain method), 73
get_unvoted_blocks() (bigchaindb.Bigchain method), 75

H

has_previous_vote() (bigchaindb.Bigchain method), 74

I

initial() (in module bigchaindb.pipelines.block), 77
initial() (in module bigchaindb.pipelines.vote), 78
is_valid_transaction() (bigchaindb.Bigchain method), 72

P

prepare_genesis_block() (bigchaindb.Bigchain method), 75

R

reassign_transaction() (bigchaindb.Bigchain method), 72

reassign_transactions() (bigchaindb.pipelines.stale.StaleTransactionMonitor method), 79
requeue_transactions() (bigchaindb.pipelines.election.Election method), 78

S

StaleTransactionMonitor (class in bigchaindb.pipelines.stale), 78
start() (in module bigchaindb.pipelines.block), 77
start() (in module bigchaindb.pipelines.stale), 79
start() (in module bigchaindb.pipelines.vote), 78

U

ungroup() (bigchaindb.pipelines.vote.Vote method), 77

V

validate_block() (bigchaindb.Bigchain method), 74
validate_block() (bigchaindb.consensus.BaseConsensusRules static method), 75
validate_transaction() (bigchaindb.Bigchain method), 72
validate_transaction() (bigchaindb.consensus.BaseConsensusRules static method), 75
validate_tx() (bigchaindb.pipelines.block.BlockPipeline method), 76
validate_tx() (bigchaindb.pipelines.vote.Vote method), 77
verify_vote_signature() (bigchaindb.consensus.BaseConsensusRules static method), 75
Vote (class in bigchaindb.pipelines.vote), 77
vote() (bigchaindb.Bigchain method), 75
vote() (bigchaindb.pipelines.vote.Vote method), 77

W

write() (bigchaindb.pipelines.block.BlockPipeline method), 76
write_block() (bigchaindb.Bigchain method), 75
write_transaction() (bigchaindb.Bigchain method), 72
write_vote() (bigchaindb.Bigchain method), 75
write_vote() (bigchaindb.pipelines.vote.Vote method), 78