BigchainDB Server Documentation

Release 0.10.0.dev

BigchainDB Contributors

Mar 29, 2017

Contents

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 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 consortium's BigchainDB cluster. A production node has the most components and requirements.

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 cluster (including production nodes)

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

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 16.04 or similar. If you're not using Linux, then you might try running BigchainDB with Docker.

A. Install MongoDB as the database backend. (There are other options but you can ignore them for now.)

Install MongoDB Server 3.4+

B. Run MongoDB. Open a Terminal and run the command:

```
$ mongod --replSet=bigchain-rs
```

C. Ubuntu 16.04 already has Python 3.5, so you don't need to install it, but you do need to install some 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 BigchainDB Server:

```
$ bigchaindb -y configure mongodb
```

G. Run BigchainDB Server:

```
$ bigchaindb start
```

You now have a running BigchainDB Server and can post transactions to it. One way to do that is to use the BigchainDB Python Driver.

Install the BigchainDB Python Driver (link)

CHAPTER 3

Cloud Deployment Templates

We have some "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.

You don't have to use the tools we use in the templates. You can use whatever tools you prefer.

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

Template: Using Terraform to Provision an Ubuntu Machine on AWS

If you didn't read the introduction to the cloud deployment 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 16.04) and other resources on AWS. That machine can then be used to host a one-machine BigchainDB node.

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.

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`.

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 16.04 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.

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.

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

If you didn't read the introduction to the cloud deployment 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 16.04.

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. (Python 3 support is coming: "Ansible 2.2 features a tech preview of Python 3 support." and the latest version, as of January 31, 2017, was 2.2.1.0. For now, it's probably best to use it with Python 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
```

About Our Example Ansible Playbook

Our example Ansible playbook installs, configures and runs a basic BigchainDB node on an Ubuntu 16.04 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 16.04 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.)

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.

Run the Ansible Playbook(s)

The latest Ubuntu 16.04 AMIs from Canonical don't include Python 2 (which is required by Ansible), so the first step is to run a small Ansible playbook to install Python 2 on the managed node:

```
# cd to the directory .../bigchaindb/ntools/one-m/ansible
ansible-playbook -i hosts --private-key ~/.ssh/<key-name> install-python2.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).

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
```

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.

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

- It ensures that the installed version of RethinkDB is the latest. 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.)

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.

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.

Azure Quickstart Template

If you didn't read the introduction to the cloud deployment 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.

Note: There was an Azure quickstart template in the `blockchain` directory of Microsoft's `Azure/azure-quickstart-templates` repository on GitHub. It's gone now; it was replaced by the one described here.

One can deploy a BigchainDB node on Azure using the template in the `bigchaindb-on-ubuntu` directory of Microsoft's `Azure/azure-quickstart-templates` repository on GitHub. Here's how:

1. Go to that directory on GitHub.
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. Once you are logged in to the Microsoft Azure Portal, you should be taken to a form titled **BigchainDB**. Some notes to help with filling in that form are available below.
5. Deployment takes a few minutes. You can follow the notifications by clicking the bell icon at the top of the screen. At the time of writing, the final deployment operation (running the `init.sh` script) was failing, but a pull request (#2884) has been made to fix that and these instructions say what you can do before that pull request gets merged...
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, i.e. do `ssh <Admin_username>@<machine-ip>` where `<Admin_username>` is the admin username you entered into the form and `<machine-ip>` is the virtual machine IP address determined in the last step. Example: `ssh bcdbadmin@40.69.87.250`
8. You should be prompted for a password. Give the `<Admin_password>` you entered into the form.
9. Configure BigchainDB Server by doing:

```
bigchaindb configure rethinkdb
```

It will ask you several questions. You can press `Enter` (or `Return`) to accept the default for all of them *except for one*. When it asks **API Server bind? (default 'localhost:9984'):**, you should answer:

```
API Server bind? (default `localhost:9984`): 0.0.0.0:9984
```

Finally, run BigchainDB Server by doing:

```
bigchaindb start
```

BigchainDB Server should now be running on the Azure virtual machine.

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

Notes on the Blockchain Template Form Fields

BASICS

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.

SETTINGS

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 virtual machine.

Dns_label_prefix - Once your virtual machine 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.

Virtual_machine_size - This should be one of Azure's standard virtual machine sizes, such as `Standard_D1_v2`. There's a list of virtual machine sizes in the Azure docs.

_artifacts Location - Leave this alone.

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

TERMS AND CONDITIONS

Read the terms and conditions. If you agree to them, then check the checkbox.

Finally, click the button labelled **Purchase**. (Generally speaking, it costs money to run stuff on Azure.)

Template: Deploy a Kubernetes Cluster on Azure

A BigchainDB node can be run inside a Kubernetes cluster. This page describes one way to deploy a Kubernetes cluster on Azure.

Step 1: Get a Pay-As-You-Go Azure Subscription

Microsoft Azure has a Free Trial subscription (at the time of writing), but it's too limited to run an advanced BigchainDB node. Sign up for a Pay-As-You-Go Azure subscription via the Azure website.

You may find that you have to sign up for a Free Trial subscription first. That's okay: you can have many subscriptions.

Step 2: Create an SSH Key Pair

You'll want an SSH key pair so you'll be able to SSH to the virtual machines that you'll deploy in the next step. (If you already have an SSH key pair, you *could* reuse it, but it's probably a good idea to make a new SSH key pair for your Kubernetes VMs and nothing else.)

See the *page about how to generate a key pair for SSH*.

Step 3: Deploy an Azure Container Service (ACS)

It's *possible* to deploy an Azure Container Service (ACS) from the Azure Portal (i.e. online in your web browser) but it's actually easier to do it using the Azure Command-Line Interface (CLI).

Microsoft has instructions to install the Azure CLI 2.0 on most common operating systems. Do that.

First, update the Azure CLI to the latest version:

```
$ az component update
```

Next, login to your account using:

```
$ az login
```

It will tell you to open a web page and to copy a code to that page.

If the login is a success, you will see some information about all your subscriptions, including the one that is currently enabled (`"state": "Enabled"`). If the wrong one is enabled, you can switch to the right one using:

```
$ az account set --subscription <subscription name or ID>
```

Next, you will have to pick the Azure data center location where you'd like to deploy your cluster. You can get a list of all available locations using:

```
$ az account list-locations
```

Next, create an Azure "resource group" to contain all the resources (virtual machines, subnets, etc.) associated with your soon-to-be-deployed cluster. You can name it whatever you like but avoid fancy characters because they may confuse some software.

```
$ az group create --name <resource group name> --location <location name>
```

Example location names are `koreacentral` and `westeurope`.

Finally, you can deploy an ACS using something like:

```
$ az acs create --name <a made-up cluster name> \
--resource-group <name of resource group created earlier> \
--master-count 3 \
--agent-count 3 \
--admin-username ubuntu \
```

```
--agent-vm-size Standard_D2_v2 \
--dns-prefix <make up a name> \
--ssh-key-value ~/.ssh/<name>.pub \
--orchestrator-type kubernetes
```

There are more options. For help understanding all the options, use the built-in help:

```
$ az acs create --help
```

It takes a few minutes for all the resources to deploy. You can watch the progress in the Azure Portal: go to **Resource groups** (with the blue cube icon) and click on the one you created to see all the resources in it.

Optional: SSH to Your New Kubernetes Cluster Nodes

You can SSH to one of the just-deployed Kubernetes "master" nodes (virtual machines) using:

```
$ ssh -i ~/.ssh/<name>.pub ubuntu@<master-ip-address-or-hostname>
```

where you can get the IP address or hostname of a master node from the Azure Portal. For example:

```
$ ssh -i ~/.ssh/mykey123.pub ubuntu@mydnsprefix.westeurope.cloudapp.azure.com
```

Note: All the master nodes should have the *same* IP address and hostname (also called the Master FQDN).

The "agent" nodes shouldn't get public IP addresses or hostnames, so you can't SSH to them *directly*, but you can first SSH to the master and then SSH to an agent from there. To do that, you could copy your SSH key pair to the master (a bad idea), or use SSH agent forwarding (better). To do the latter, do the following on the machine you used to SSH to the master:

```
$ echo -e "Host <FQDN of the cluster from Azure Portal>\n  ForwardAgent yes" >> ~/.
↪ssh/config
```

To verify that SSH agent forwarding works properly, SSH to the one of the master nodes and do:

```
$ echo "$SSH_AUTH_SOCK"
```

If you get an empty response, then SSH agent forwarding hasn't been set up correctly. If you get a non-empty response, then SSH agent forwarding should work fine and you can SSH to one of the agent nodes (from a master) using something like:

```
$ ssh ubuntu@k8s-agent-4AC80E97-0
```

where `k8s-agent-4AC80E97-0` is the name of a Kubernetes agent node in your Kubernetes cluster. You will have to replace it by the name of an agent node in your cluster.

Optional: Delete the Kubernetes Cluster

```
$ az acs delete \
--name <ACS cluster name> \
--resource-group <name of resource group containing the cluster>
```

Optional: Delete the Resource Group

CAUTION: You might end up deleting resources other than the ACS cluster.

```
$ az group delete \
--name <name of resource group containing the cluster>
```

Next, you can *run a BigchainDB node on your new Kubernetes cluster*.

Kubernetes Template: Deploy a Single BigchainDB Node

This page describes how to deploy the first BigchainDB node in a BigchainDB cluster, or a stand-alone BigchainDB node, using Kubernetes. It assumes you already have a running Kubernetes cluster.

If you want to add a new BigchainDB node to an existing BigchainDB cluster, refer to *the page about that*.

Step 1: Install kubectl

kubectl is the Kubernetes CLI. If you don't already have it installed, then see the Kubernetes docs to install it.

Step 2: Configure kubectl

The default location of the kubectl configuration file is `~/.kube/config`. If you don't have that file, then you need to get it.

Azure. If you deployed your Kubernetes cluster on Azure using the Azure CLI 2.0 (as per *our template*), then you can get the `~/.kube/config` file using:

```
$ az acs kubernetes get-credentials \
--resource-group <name of resource group containing the cluster> \
--name <ACS cluster name>
```

If it asks for a password (to unlock the SSH key) and you enter the correct password, but you get an error message, then try adding `--ssh-key-file ~/.ssh/<name>` to the above command (i.e. the path to the private key).

Step 3: Create Storage Classes

MongoDB needs somewhere to store its data persistently, outside the container where MongoDB is running. Our MongoDB Docker container (based on the official MongoDB Docker container) exports two volume mounts with correct permissions from inside the container:

- The directory where the mongod instance stores its data: `/data/db`. There's more explanation in the MongoDB docs about storage.dbpath.
- The directory where the mongodb instance stores the metadata for a sharded cluster: `/data/configdb/`. There's more explanation in the MongoDB docs about sharding.configDB.

Explaining how Kubernetes handles persistent volumes, and the associated terminology, is beyond the scope of this documentation; see the Kubernetes docs about persistent volumes.

The first thing to do is create the Kubernetes storage classes.

Azure. First, you need an Azure storage account. If you deployed your Kubernetes cluster on Azure using the Azure CLI 2.0 (as per *our template*), then the *az acs create* command already created two storage accounts in the

same location and resource group as your Kubernetes cluster. Both should have the same "storage account SKU": `Standard_LRS`. Standard storage is lower-cost and lower-performance. It uses hard disk drives (HDD). LRS means locally-redundant storage: three replicas in the same data center. Premium storage is higher-cost and higher-performance. It uses solid state drives (SSD). At the time of writing, when we created a storage account with SKU `Premium_LRS` and tried to use that, the PersistentVolumeClaim would get stuck in a "Pending" state. For future reference, the command to create a storage account is az storage account create.

Get the file `mongo-sc.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/mongodb/
↪mongo-sc.yaml
```

You may have to update the `parameters.location` field in both the files to specify the location you are using in Azure.

Create the required storage classes using:

```
$ kubectl apply -f mongo-sc.yaml
```

You can check if it worked using `kubectl get storageclasses`.

Azure. Note that there is no line of the form `storageAccount: <azure storage account name>` under `parameters:`. When we included one and then created a PersistentVolumeClaim based on it, the PersistentVolumeClaim would get stuck in a "Pending" state. Kubernetes just looks for a storageAccount with the specified skuName and location.

Step 4: Create Persistent Volume Claims

Next, you will create two PersistentVolumeClaim objects `mongo-db-claim` and `mongo-configdb-claim`. Get the file `mongo-pvc.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/mongodb/
↪mongo-pvc.yaml
```

Note how there's no explicit mention of Azure, AWS or whatever. `ReadWriteOnce` (RWO) means the volume can be mounted as read-write by a single Kubernetes node. (`ReadWriteOnce` is the *only* access mode supported by AzureDisk.) `storage: 20Gi` means the volume has a size of 20 gibibytes.

You may want to update the `spec.resources.requests.storage` field in both the files to specify a different disk size.

Create the required Persistent Volume Claims using:

```
$ kubectl apply -f mongo-pvc.yaml
```

You can check its status using: `kubectl get pvc -w`

Initially, the status of persistent volume claims might be "Pending" but it should become "Bound" fairly quickly.

Step 5: Create the Config Map - Optional

This step is required only if you are planning to set up multiple BigchainDB nodes.

MongoDB reads the local `/etc/hosts` file while bootstrapping a replica set to resolve the hostname provided to the `rs.initiate()` command. It needs to ensure that the replica set is being initialized in the same instance where the MongoDB instance is running.

To achieve this, you will create a ConfigMap with the FQDN of the MongoDB instance and populate the `/etc/hosts` file with this value so that a replica set can be created seamlessly.

Get the file `mongo-cm.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/mongodb/
↪mongo-cm.yaml
```

You may want to update the `data.fqdn` field in the file before creating the ConfigMap. `data.fqdn` field will be the DNS name of your MongoDB instance. This will be used by other MongoDB instances when forming a MongoDB replica set. It should resolve to the MongoDB instance in your cluster when you are done with the setup. This will help when you are adding more MongoDB instances to the replica set in the future.

Azure. In Kubernetes on ACS, the name you populate in the `data.fqdn` field will be used to configure a DNS name for the public IP assigned to the Kubernetes Service that is the frontend for the MongoDB instance. We suggest using a name that will already be available in Azure. We use `mdb-instance-0`, `mdb-instance-1` and so on in this document, which gives us `mdb-instance-0.<azure location>.cloudapp.azure.com`, `mdb-instance-1.<azure location>.cloudapp.azure.com`, etc. as the FQDNs. The `<azure location>` is the Azure datacenter location you are using, which can also be obtained using the `az account list-locations` command. You can also try to assign a name to an Public IP in Azure before starting the process, or use `nslookup` with the name you have in mind to check if it's available for use.

You should ensure that the the name specified in the `data.fqdn` field is a unique one.

Kubernetes on bare-metal or other cloud providers. You need to provide the name resolution function by other means (using DNS providers like GoDaddy, CloudFlare or your own private DNS server). The DNS set up for other environments is currently beyond the scope of this document.

Create the required ConfigMap using:

```
$ kubectl apply -f mongo-cm.yaml
```

You can check its status using: `kubectl get cm`

Now you are ready to run MongoDB and BigchainDB on our Kubernetes cluster.

Step 6: Run MongoDB as a StatefulSet

Get the file `mongo-ss.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/mongodb/
↪mongo-ss.yaml
```

Note how the MongoDB container uses the `mongo-db-claim` and the `mongo-configdb-claim` PersistentVolumeClaims for its `/data/db` and `/data/configdb` diretories (mount path). Note also that we use the pod's `securityContext.capabilities.add` specification to add the `FOWNER` capability to the container. That is because MongoDB container has the user `mongodb`, with uid `999` and group `mongodb`, with gid `999`. When this container runs on a host with a mounted disk, the writes fail when there is no user with uid `999`. To avoid this, we use the Docker feature of `--cap-add=FOWNER`. This bypasses the uid and gid permission checks during writes and allows data to be persisted to disk. Refer to the Docker docs for details.

As we gain more experience running MongoDB in testing and production, we will tweak the `resources.limits.cpu` and `resources.limits.memory`. We will also stop exposing port `27017` globally and/or allow only certain hosts to connect to the MongoDB instance in the future.

Create the required StatefulSet using:

```
$ kubectl apply -f mongo-ss.yaml
```

You can check its status using the commands `kubectl get statefulsets -w` and `kubectl get svc -w`

You may have to wait for up to 10 minutes for the disk to be created and attached on the first run. The pod can fail several times with the message saying that the timeout for mounting the disk was exceeded.

Step 7: Initialize a MongoDB Replica Set - Optional

This step is required only if you are planning to set up multiple BigchainDB nodes.

Login to the running MongoDB instance and access the mongo shell using:

```
$ kubectl exec -it mdb-0 -c mongodb -- /bin/bash
root@mdb-0:/# mongo --port 27017
```

You will initiate the replica set by using the `rs.initiate()` command from the mongo shell. Its syntax is:

```
rs.initiate({
    _id : "<replica-set-name",
    members: [ {
      _id : 0,
      host : "<fqdn of this instance>:<port number>"
    } ]
})
```

An example command might look like:

```
> rs.initiate({ _id : "bigchain-rs", members: [ { _id : 0, host :"mdb-instance-0.
↪westeurope.cloudapp.azure.com:27017" } ] })
```

where `mdb-instance-0.westeurope.cloudapp.azure.com` is the value stored in the `data.fqdn` field in the ConfigMap created using `mongo-cm.yaml`.

You should see changes in the mongo shell prompt from `>` to `bigchain-rs:OTHER>` to `bigchain-rs:SECONDARY>` and finally to `bigchain-rs:PRIMARY>`.

You can use the `rs.conf()` and the `rs.status()` commands to check the detailed replica set configuration now.

Step 8: Create a DNS record - Optional

This step is required only if you are planning to set up multiple BigchainDB nodes.

Azure. Select the current Azure resource group and look for the `Public IP` resource. You should see at least 2 entries there - one for the Kubernetes master and the other for the MongoDB instance. You may have to `Refresh` the Azure web page listing the resources in a resource group for the latest changes to be reflected. Select the `Public IP` resource that is attached to your service (it should have the Kubernetes cluster name along with a random string), select `Configuration`, add the DNS name that was added in the ConfigMap earlier, click `Save`, and wait for the changes to be applied.

To verify the DNS setting is operational, you can run `nslookup <dns name added in ConfigMap>` from your local Linux shell.

This will ensure that when you scale the replica set later, other MongoDB members in the replica set can reach this instance.

Step 9: Run BigchainDB as a Deployment

Get the file `bigchaindb-dep.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/bigchaindb/
↪bigchaindb-dep.yaml
```

Note that we set the `BIGCHAINDB_DATABASE_HOST` to `mdb-svc` which is the name of the MongoDB service defined earlier.

We also hardcode the `BIGCHAINDB_KEYPAIR_PUBLIC`, `BIGCHAINDB_KEYPAIR_PRIVATE` and `BIGCHAINDB_KEYRING` for now.

As we gain more experience running BigchainDB in testing and production, we will tweak the `resources.limits` values for CPU and memory, and as richer monitoring and probing becomes available in BigchainDB, we will tweak the `livenessProbe` and `readinessProbe` parameters.

We also plan to specify scheduling policies for the BigchainDB deployment so that we ensure that BigchainDB and MongoDB are running in separate nodes, and build security around the globally exposed port `9984`.

Create the required Deployment using:

```
$ kubectl apply -f bigchaindb-dep.yaml
```

You can check its status using the command `kubectl get deploy -w`

Step 10: Run NGINX as a Deployment

NGINX is used as a proxy to both the BigchainDB and MongoDB instances in the node. It proxies HTTP requests on port 80 to the BigchainDB backend, and TCP connections on port 27017 to the MongoDB backend.

You can also configure a whitelist in NGINX to allow only connections from other instances in the MongoDB replica set to access the backend MongoDB instance.

Get the file `nginx-cm.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/nginx/nginx-
↪cm.yaml
```

The IP address whitelist can be explicitly configured in `nginx-cm.yaml` file. You will need a list of the IP addresses of all the other MongoDB instances in the cluster. If the MongoDB intances specify a hostname, then this needs to be resolved to the corresponding IP addresses. If the IP address of any MongoDB instance changes, we can start a 'rolling upgrade' of NGINX after updating the corresponding ConfigMap without affecting availabilty.

Create the ConfigMap for the whitelist using:

```
$ kubectl apply -f nginx-cm.yaml
```

Get the file `nginx-dep.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/nginx/nginx-
↪dep.yaml
```

Create the NGINX deployment using:

```
$ kubectl apply -f nginx-dep.yaml
```

Step 11: Verify the BigchainDB Node Setup

Step 11.1: Testing Internally

Run a container that provides utilities like `nslookup`, `curl` and `dig` on the cluster and query the internal DNS and IP endpoints.

```
$ kubectl run -it toolbox -- image <docker image to run> --restart=Never --rm
```

There is a generic image based on alpine:3.5 with the required utilities hosted at Docker Hub under `bigchaindb/toolbox`. The corresponding Dockerfile is here.

You can use it as below to get started immediately:

```
$ kubectl run -it toolbox --image bigchaindb/toolbox --restart=Never --rm
```

It will drop you to the shell prompt. Now you can query for the `mdb` and `bdb` service details.

```
# nslookup mdb-svc
# nslookup bdb-svc
# nslookup ngx-svc
# dig +noall +answer _mdb-port._tcp.mdb-svc.default.svc.cluster.local SRV
# dig +noall +answer _bdb-port._tcp.bdb-svc.default.svc.cluster.local SRV
# dig +noall +answer _ngx-public-mdb-port._tcp.ngx-svc.default.svc.cluster.local SRV
# dig +noall +answer _ngx-public-bdb-port._tcp.ngx-svc.default.svc.cluster.local SRV
# curl -X GET http://mdb-svc:27017
# curl -X GET http://bdb-svc:9984
# curl -X GET http://ngx-svc:80
# curl -X GET http://ngx-svc:27017
```

The `nslookup` commands should output the configured IP addresses of the services in the cluster

The `dig` commands should return the port numbers configured for the various services in the cluster.

Finally, the `curl` commands test the availability of the services themselves.

Step 11.2: Testing Externally

Try to access the `<dns/ip of your exposed bigchaindb service endpoint>:80` on your browser. You must receive a json output that shows the BigchainDB server version among other things.

Try to access the `<dns/ip of your exposed mongodb service endpoint>:27017` on your browser. If your IP is in the whitelist, you will receive a message from the MongoDB instance stating that it doesn't allow HTTP connections to the port anymore. If your IP is not in the whitelist, your access will be blocked and you will not see any response from the MongoDB instance.

Kubernetes Template: Add a BigchainDB Node to an Existing BigchainDB Cluster

This page describes how to deploy a BigchainDB node using Kubernetes, and how to add that node to an existing BigchainDB cluster. It assumes you already have a running Kubernetes cluster where you can deploy the new BigchainDB node.

If you want to deploy the first BigchainDB node in a BigchainDB cluster, or a stand-alone BigchainDB node, then see *the page about that*.

Terminology Used

`existing cluster` will refer to one of the existing Kubernetes clusters hosting one of the existing BigchainDB nodes.

`ctx-1` will refer to the kubectl context of the existing cluster.

`new cluster` will refer to the new Kubernetes cluster that will run a new BigchainDB node (including a BigchainDB instance and a MongoDB instance).

`ctx-2` will refer to the kubectl context of the new cluster.

`new MongoDB instance` will refer to the MongoDB instance in the new cluster.

`existing MongoDB instance` will refer to the MongoDB instance in the existing cluster.

`new BigchainDB instance` will refer to the BigchainDB instance in the new cluster.

`existing BigchainDB instance` will refer to the BigchainDB instance in the existing cluster.

Step 1: Prerequisites

- A public/private key pair for the new BigchainDB instance.
- The public key should be shared offline with the other existing BigchainDB nodes in the existing BigchainDB cluster.
- You will need the public keys of all the existing BigchainDB nodes.
- A new Kubernetes cluster setup with kubectl configured to access it.
- Some familiarity with deploying a BigchainDB node on Kubernetes. See our *other docs about that*.

Note: If you are managing multiple Kubernetes clusters, from your local system, you can run `kubectl config view` to list all the contexts that are available for the local kubectl. To target a specific cluster, add a `--context` flag to the kubectl CLI. For example:

```
$ kubectl --context ctx-1 apply -f example.yaml
$ kubectl --context ctx-2 apply -f example.yaml
$ kubectl --context ctx-1 proxy --port 8001
$ kubectl --context ctx-2 proxy --port 8002
```

Step 2: Prepare the New Kubernetes Cluster

Follow the steps in the sections to set up Storage Classes and Persistent Volume Claims, and to run MongoDB in the new cluster:

1. *Add Storage Classes*
2. *Add Persistent Volume Claims*
3. *Create the Config Map*
4. *Run MongoDB instance*

Step 3: Add the New MongoDB Instance to the Existing Replica Set

Note that by `replica set`, we are referring to the MongoDB replica set, not a Kubernetes' `ReplicaSet`.

If you are not the administrator of an existing BigchainDB node, you will have to coordinate offline with an existing administrator so that they can add the new MongoDB instance to the replica set.

Add the new instance of MongoDB from an existing instance by accessing the `mongo` shell.

```
$ kubectl --context ctx-1 exec -it mdb-0 -c mongodb -- /bin/bash
root@mdb-0# mongo --port 27017
```

One can only add members to a replica set from the `PRIMARY` instance. The `mongo` shell prompt should state that this is the primary member in the replica set. If not, then you can use the `rs.status()` command to find out who the primary is and login to the `mongo` shell in the primary.

Run the `rs.add()` command with the FQDN and port number of the other instances:

```
PRIMARY> rs.add("<fqdn>:<port>")
```

Step 4: Verify the Replica Set Membership

You can use the `rs.conf()` and the `rs.status()` commands available in the mongo shell to verify the replica set membership.

The new MongoDB instance should be listed in the membership information displayed.

Step 5: Start the New BigchainDB Instance

Get the file `bigchaindb-dep.yaml` from GitHub using:

```
$ wget https://raw.githubusercontent.com/bigchaindb/bigchaindb/master/k8s/bigchaindb/
↪bigchaindb-dep.yaml
```

Note that we set the `BIGCHAINDB_DATABASE_HOST` to `mdb` which is the name of the MongoDB service defined earlier.

Edit the `BIGCHAINDB_KEYPAIR_PUBLIC` with the public key of this instance, the `BIGCHAINDB_KEYPAIR_PRIVATE` with the private key of this instance and the `BIGCHAINDB_KEYRING` with a `:` delimited list of all the public keys in the BigchainDB cluster.

Create the required Deployment using:

```
$ kubectl --context ctx-2 apply -f bigchaindb-dep.yaml
```

You can check its status using the command `kubectl get deploy -w`

Step 6: Restart the Existing BigchainDB Instance(s)

Add the public key of the new BigchainDB instance to the keyring of all the existing BigchainDB instances and update the BigchainDB instances using:

```
$ kubectl --context ctx-1 replace -f bigchaindb-dep.yaml
```

This will create a "rolling deployment" in Kubernetes where a new instance of BigchainDB will be created, and if the health check on the new instance is successful, the earlier one will be terminated. This ensures that there is zero downtime during updates.

You can SSH to an existing BigchainDB instance and run the `bigchaindb show-config` command to check that the keyring is updated.

Step 7: Run NGINX as a Deployment

Please refer *this* to set up NGINX in your new node.

Step 8: Test Your New BigchainDB Node

Please refer to the testing steps *here* to verify that your new BigchainDB node is working as expected.

Kubernetes Template: Upgrade all Software in a BigchainDB Node

This page outlines how to upgrade all the software associated with a BigchainDB node running on Kubernetes, including host operating systems, Docker, Kubernetes, and BigchainDB-related software.

Upgrade Host OS, Docker and Kubernetes

Some Kubernetes installation & management systems can do full or partial upgrades of host OSes, Docker, or Kubernetes, e.g. Tectonic, Rancher, and Kubo. Consult the documentation for your system.

Azure Container Service (ACS). On Dec. 15, 2016, a Microsoft employee wrote: "In the coming months we [the Azure Kubernetes team] will be building managed updates in the ACS service." At the time of writing, managed updates were not yet available, but you should check the latest ACS documentation to see what's available now. Also at the time of writing, ACS only supported Ubuntu as the host (master and agent) operating system. You can upgrade Ubuntu and Docker on Azure by SSHing into each of the hosts, as documented on *another page*.

In general, you can SSH to each host in your Kubernetes Cluster to update the OS and Docker.

Note: Once you are in an SSH session with a host, the `docker info` command is a handy way to detemine the host OS (including version) and the Docker version.

When you want to upgrade the software on a Kubernetes node, you should "drain" the node first, i.e. tell Kubernetes to gracefully terminate all pods on the node and mark it as unscheduleable (so no new pods get put on the node during its downtime).

```
kubectl drain $NODENAME
```

There are more details in the Kubernetes docs, including instructions to make the node scheduleable again.

To manually upgrade the host OS, see the docs for that OS.

To manually upgrade Docker, see the Docker docs.

To manually upgrade all Kubernetes software in your Kubernetes cluster, see the Kubernetes docs.

Upgrade BigchainDB-Related Software

We use Kubernetes "Deployments" for NGINX, BigchainDB, and most other BigchainDB-related software. The only exception is MongoDB; we use a Kubernetes StatefulSet for that.

The nice thing about Kubernetes Deployments is that Kubernetes can manage most of the upgrade process. A typical upgrade workflow for a single Deployment would be:

```
$ KUBE_EDITOR=nano kubectl edit deployment/<name of Deployment>
```

The kubectl edit command opens the specified editor (nano in the above example), allowing you to edit the specified Deployment *in the Kubernetes cluster*. You can change the version tag on the Docker image, for example. Don't forget to save your edits before exiting the editor. The Kubernetes docs have more information about updating a Deployment.

The upgrade story for the MongoDB StatefulSet is *different*. (This is because MongoDB has persistent state, which is stored in some storage associated with a PersistentVolumeClaim.) At the time of writing, StatefulSets were still in beta, and they did not support automated image upgrade (Docker image tag upgrade). We expect that to change. Rather than trying to keep these docs up-to-date, we advise you to check out the current Kubernetes docs about updating containers in StatefulSets.

CHAPTER 4

Production Node Assumptions, Components & Requirements

Production Node Assumptions

If you're not sure what we mean by a BigchainDB *node*, *cluster*, *consortium*, 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 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 templates, but those are just a starting point.)

Production Node Components

A BigchainDB node must include, at least:

- BigchainDB Server and
- RethinkDB Server.

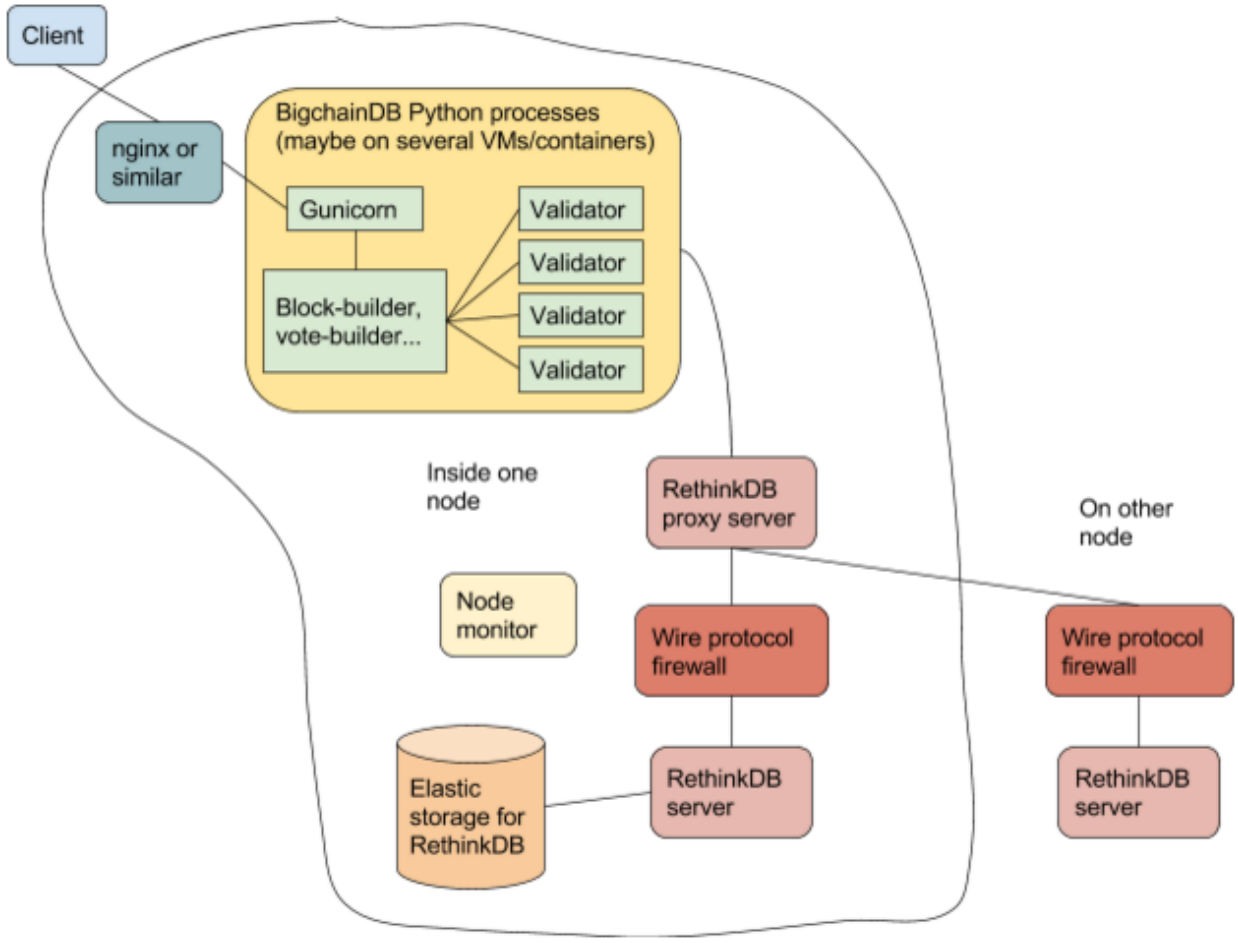
When doing development and testing, it's common to install both on the same machine, but in a production environment, it may make more sense to install them on separate machines.

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.



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.

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 recent LTS versions of Ubuntu and recent versions of Fedora.

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.

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.

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).

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.

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.

Get a Server

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

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.

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 cluster 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.

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`.

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.

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 cluster
```

- `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.

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.

Install BigchainDB Server

First, install the OS-level dependencies of BigchainDB Server (link).

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 16.04, 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
```

Configure BigchainDB Server

Start by creating a default BigchainDB config file:

```
bigchaindb -y configure rethinkdb
```

(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 `"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 cluster. The keyring should *not* include your node's public key.

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

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/.

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 consortium. 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

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).

Option A: Using a Local Dev Machine

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

With RethinkDB

Create a default BigchainDB config file (in `$HOME/.bigchaindb`):

```
$ bigchaindb -y configure rethinkdb
```

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.

With MongoDB

Create a default BigchainDB config file (in `$HOME/.bigchaindb`):

```
$ bigchaindb -y configure mongodb
```

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

Start MongoDB **3.4+** using:

```
$ mongod --replSet=bigchain-rs
```

You can verify that MongoDB is running correctly by checking the output of the previous command for the line:

```
waiting for connections on port 27017
```

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.

Option B: 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
```

Docker with RethinkDB

Note: If you're upgrading BigchainDB and have previously already built the images, you may need to rebuild them after the upgrade to install any new dependencies.

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 access 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
```

Docker with MongoDB

Start MongoDB:

```
docker-compose up -d mdb
```

MongoDB should now be up and running. You can check the port binding for the MongoDB driver port using:

```
$ docker-compose port mdb 27017
```

Start a BigchainDB node:

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

You can monitor the logs:

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

If you wish to run the tests:

```
docker-compose run --rm bdb-mdb py.test -v --database-backend=mongodb
```

Accessing the HTTP API

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

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

should give you something like:

```
{
  "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
```

Option C: 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

Running All Tests

All documentation about writing and running tests (unit and integration tests) was moved to the file `bigchaindb/tests/README.md`.

CHAPTER 6

Settings & CLI

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_BACKEND  BIGCHAINDB_DATABASE_HOST  BIGCHAINDB_DATABASE_PORT
BIGCHAINDB_DATABASE_NAME BIGCHAINDB_DATABASE_REPLICASET BIGCHAINDB_SERVER_BIND
BIGCHAINDB_SERVER_WORKERS   BIGCHAINDB_SERVER_THREADS   BIGCHAINDB_CONFIG_PATH
BIGCHAINDB_BACKLOG_REASSIGN_DELAY                    BIGCHAINDB_CONSENSUS_PLUGIN
BIGCHAINDB_LOG         BIGCHAINDB_LOG_FILE         BIGCHAINDB_LOG_LEVEL_CONSOLE
BIGCHAINDB_LOG_LEVEL_LOGFILE                      BIGCHAINDB_LOG_DATEFMT_CONSOLE
BIGCHAINDB_LOG_DATEFMT_LOGFILE                       BIGCHAINDB_LOG_FMT_CONSOLE
BIGCHAINDB_LOG_FMT_LOGFILE BIGCHAINDB_LOG_GRANULAR_LEVELS
```

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 rethinkdb` 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`).

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 rethinkdb`.

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": []
```

database.backend, database.host, database.port, database.name & database.replicaset

The database backend to use (`rethinkdb` or `mongodb`) and its hostname, port and name. If the database backend is `mongodb`, then there's a fifth setting: the name of the replica set. If the database backend is `rethinkdb`, you *can* set the name of the replica set, but it won't be used for anything.

Example using environment variables

```
export BIGCHAINDB_DATABASE_BACKEND=mongodb
export BIGCHAINDB_DATABASE_HOST=localhost
export BIGCHAINDB_DATABASE_PORT=27017
```

```
export BIGCHAINDB_DATABASE_NAME=bigchain
export BIGCHAINDB_DATABASE_REPLICASET=bigchain-rs
```

Default values

If (no environment variables were set and there's no local config file), or you used `bigchaindb -y configure rethinkdb` to create a default local config file for a RethinkDB backend, then the defaults will be:

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

If you used `bigchaindb -y configure mongodb` to create a default local config file for a MongoDB backend, then the defaults will be:

```
"database": {
    "backend": "mongodb",
    "host": "localhost",
    "name": "bigchain",
    "port": 27017,
    "replicaset": "bigchain-rs"
}
```

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
}
```

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
```

consensus_plugin

The consensus plugin to use.

Example using an environment variable

```
export BIGCHAINDB_CONSENSUS_PLUGIN=default
```

Example config file snippet: the default

```
"consensus_plugin": "default"
```

log

The `log` key is expected to point to a mapping (set of key/value pairs) holding the logging configuration.

Example:

```
{
    "log": {
        "file": "/var/log/bigchaindb.log",
        "level_console": "info",
        "level_logfile": "info",
        "datefmt_console": "%Y-%m-%d %H:%M:%S",
        "datefmt_logfile": "%Y-%m-%d %H:%M:%S",
        "fmt_console": "%(asctime)s [%(levelname)s] (%(name)s) %(message)s",
        "fmt_logfile": "%(asctime)s [%(levelname)s] (%(name)s) %(message)s",
        "granular_levels": {
            "bichaindb.backend": "info",
            "bichaindb.core": "info"
```

```
        }
}
```

Defaults to: `"{}"`.

Please note that although the default is `"{}"` as per the configuration file, internal defaults are used, such that the actual operational default is:

```
{
    "log": {
        "file": "~/bigchaindb.log",
        "level_console": "info",
        "level_logfile": "info",
        "datefmt_console": "%Y-%m-%d %H:%M:%S",
        "datefmt_logfile": "%Y-%m-%d %H:%M:%S",
        "fmt_console": "%(asctime)s [%(levelname)s] (%(name)s) %(message)s",
        "fmt_logfile": "%(asctime)s [%(levelname)s] (%(name)s) %(message)s",
        "granular_levels": {}
}
```

The next subsections explain each field of the `log` configuration.

log.file

The full path to the file where logs should be written to.

Example:

```
{
    "log": {
        "file": "/var/log/bigchaindb/bigchaindb.log"
    }
}
```

Defaults to: `"~/bigchaindb.log"`.

Please note that the user running `bigchaindb` must have write access to the location.

log.level_console

The log level used to log to the console. Possible allowed values are the ones defined by Python, but case insensitive for convenience's sake:

```
"critical", "error", "warning", "info", "debug", "notset"
```

Example:

```
{
    "log": {
        "level_console": "info"
    }
}
```

Defaults to: `"info"`.

log.level_logfile

The log level used to log to the log file. Possible allowed values are the ones defined by Python, but case insensitive for convenience's sake:

```
"critical", "error", "warning", "info", "debug", "notset"
```

Example:

```
{
    "log": {
        "level_file": "info"
    }
}
```

Defaults to: `"info"`.

log.datefmt_console

The format string for the date/time portion of a message, when logged to the console.

Example:

```
{
    "log": {
        "datefmt_console": "%x %X %Z"
    }
}
```

Defaults to: `"%Y-%m-%d %H:%M:%S"`.

For more information on how to construct the format string please consult the table under Python's documentation of `time.strftime(format[, t])`

log.datefmt_logfile

The format string for the date/time portion of a message, when logged to a log file.

Example:

```
{
    "log": {
        "datefmt_logfile": "%c %z"
    }
}
```

Defaults to: `"%Y-%m-%d %H:%M:%S"`.

For more information on how to construct the format string please consult the table under Python's documentation of `time.strftime(format[, t])`

log.fmt_console

A string used to format the log messages when logged to the console.

Example:

```
{
    "log": {
        "fmt_console": "%(asctime)s [%(levelname)s] %(message)s %(process)d"
    }
}
```

Defaults to: `"[%(asctime)s] [%(levelname)s] (%(name)s) %(message)s (%(processName)-10s - pid: %(process)d)"`

For more information on possible formatting options please consult Python's documentation on LogRecord attributes

log.fmt_logfile

A string used to format the log messages when logged to a log file.

Example:

```
{
    "log": {
        "fmt_logfile": "%(asctime)s [%(levelname)s] %(message)s %(process)d"
    }
}
```

Defaults to: `"[%(asctime)s] [%(levelname)s] (%(name)s) %(message)s (%(processName)-10s - pid: %(process)d)"`

For more information on possible formatting options please consult Python's documentation on LogRecord attributes

log.granular_levels

Log levels for BigchainDB's modules. This can be useful to control the log level of specific parts of the application. As an example, if you wanted the logging of the `core.py` module to be more verbose, you would set the configuration shown in the example below.

Example:

```
{
    "log": {
        "granular_levels": {
            "bichaindb.core": "debug"
        }
}
```

Defaults to: `"{}"`

Command Line Interface (CLI)

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

bigchaindb –help

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

bigchaindb –version

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

bigchaindb configure

Generate a local configuration 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.

Since BigchainDB supports multiple databases you need to always specify the database backend that you want to use. At this point only two database backends are supported: `rethinkdb` and `mongodb`.

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 rethinkdb
```

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 rethinkdb
```

bigchaindb show-config

Show the values of the BigchainDB node configuration settings.

bigchaindb export-my-pubkey

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

bigchaindb init

Create a backend database (RethinkDB or MongoDB), all database tables/collections, various backend 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 backend 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).

bigchaindb drop

Drop (erase) the backend database (a RethinkDB or MongoDB 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`

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.

Options

The log level for the console can be set via the option `--log-level` or its abbreviation `-l`. Example:

```
$ bigchaindb --log-level INFO start
```

The allowed levels are `DEBUG`, `INFO` , `WARNING`, `ERROR`, and `CRITICAL`. For an explanation regarding these levels please consult the Logging Levels section of Python's documentation.

For a more fine-grained control over the logging configuration you can use the configuration file as documented under Configuration Settings.

bigchaindb set-shards

This command is specific to RethinkDB so it will only run if BigchainDB is configured with `rethinkdb` as the backend.

If RethinkDB is the backend database, then:

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

will set the number of shards (in all RethinkDB tables) to 4.

bigchaindb set-replicas

This command is specific to RethinkDB so it will only run if BigchainDB is configured with `rethinkdb` as the backend.

If RethinkDB is the backend database, then:

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

will set the number of replicas (of each shard) to 3 (i.e. it will set the replication factor to 3).

bigchaindb add-replicas

This command is specific to MongoDB so it will only run if BigchainDB is configured with `mongodb` as the backend.

This command is used to add nodes to a BigchainDB cluster. It accepts a list of space separated hosts in the form *hostname:port*:

```
$ bigchaindb add-replicas server1.com:27017 server2.com:27017 server3.com:27017
```

bigchaindb remove-replicas

This command is specific to MongoDB so it will only run if BigchainDB is configured with `mongodb` as the backend.

This command is used to remove nodes from a BigchainDB cluster. It accepts a list of space separated hosts in the form *hostname:port*:

```
$ bigchaindb remove-replicas server1.com:27017 server2.com:27017 server3.com:27017
```

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.

If you use a language other than Python, you may want to look at the current community projects listed below.

The HTTP Client-Server API

This page assumes you already know an API Root URL for a BigchainDB node or reverse proxy. It should be something like `https://example.com:9984` or `https://12.34.56.78:9984`.

If you set up a BigchainDB node or reverse proxy yourself, and you're not sure what the API Root URL is, then see the last section of this page for help.

BigchainDB Root URL

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

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

{
  "_links": {
    "api_v1": "http://example.com:9984/api/v1/",
    "docs": "https://docs.bigchaindb.com/projects/server/en/v0.10.0.dev/"
  },
  "keyring": [
    "6qHyZew94NMmUTYyHnkZsB8cxJYuRNEiEpXHe1ih9QX3",
    "AdDuyrTyjrDt935YnFu4VBCVDhHtY2Y6rcy7x2TFeiRi"
```

```
  ],
  "public_key": "NC8c8rYcAhyKVpx1PCV65CBmyq4YUbLysy3Rqrg8L8mz",
  "software": "BigchainDB",
  "version": "0.10.0.dev"
}
```

API Root Endpoint

If you send an HTTP GET request to the API Root Endpoint e.g. `http://localhost:9984/api/v1/` or `https://example.com:9984/api/v1/`, then you should get an HTTP response that allows you to discover the BigchainDB API endpoints:

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

{
  "_links": {
    "docs": "https://docs.bigchaindb.com/projects/server/en/v0.10.0.dev/drivers-
↪clients/http-client-server-api.html",
    "self": "http://example.com:9984/api/v1/",
    "statuses": "http://example.com:9984/api/v1/statuses/",
    "transactions": "http://example.com:9984/api/v1/transactions/"
  }
}
```

Transactions

GET /api/v1/transactions/{tx_id}

Get the transaction with the ID `tx_id`.

This endpoint returns a transaction only if a `VALID` block on `bigchain` exists.

Parameters

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

Example request:

```
GET /api/v1/transactions/
↪04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd HTTP/1.1
Host: example.com
```

Example response:

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

{
  "asset": {
    "data": {
      "msg": "Hello BigchainDB!"
    }
  },
  "id": "04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd",
  "inputs": [
    {
```

```
      "fulfillment": "cf:4:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5waJ_
↪J9gSpMu3q4p4VdDO8BMroPZTaI01B2eWbvR59_yr-WZOjxVDfmrtLGSfiqsYEPdaeTS3KAMZ2Mt_
↪jv8AMH",
      "fulfills": null,
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 0
  },
  "operation": "CREATE",
  "outputs": [
    {
      "amount": 1,
      "condition": {
        "details": {
          "bitmask": 32,
          "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
          "signature": null,
          "type": "fulfillment",
          "type_id": 4
        },
        "uri": "cc:4:20:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5w:96"
      },
      "public_keys": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "version": "0.10"
}
```


Response Headers

- Content-Type – `application/json`

Status Codes

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

GET /api/v1/transactions

The unfiltered `/api/v1/transactions` endpoint without any query parameters returns a status code *400*. For valid filters, see the sections below.

There are however filtered requests that might come of use, given the endpoint is queried correctly. Some of them include retrieving a list of transactions that include:

•*Transactions related to a specific asset*

In this section, we've listed those particular requests, as they will likely to be very handy when implementing your application on top of BigchainDB.

Note: Looking up transactions with a specific `metadata` field is currently not supported, however, providing a way to query based on `metadata` data is on our roadmap.

A generalization of those parameters follows:

Query Parameters

- **asset_id** (*string*) – The ID of the asset.
- **operation** (*string*) – (Optional) One of the two supported operations of a transaction: `CREATE`, `TRANSFER`.

GET /api/v1/transactions?asset_id={asset_id}&operation={CREATE|TRANSFER}

Get a list of transactions that use an asset with the ID `asset_id`. Every `TRANSFER` transaction that originates from a `CREATE` transaction with `asset_id` will be included. This allows users to query the entire history or provenance of an asset.

This endpoint returns transactions only if they are decided `VALID` by the server.

Query Parameters

- **operation** (*string*) – (Optional) One of the two supported operations of a transaction: `CREATE`, `TRANSFER`.
- **asset_id** (*string*) – asset ID.

Example request:

```
GET /api/v1/transactions?operation=TRANSFER&asset_
↪id=04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd HTTP/1.1
Host: example.com
```

Example response:

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

[{
  "asset": {
    "id": "04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd"
  },
  "id": "7eb94bc4114e221f9d9f24f5930c9bd3722b4e226ffc267e40fd3739a2a09670",
  "inputs": [
    {
      "fulfillment": "cf:4:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5w6if-
↪1q8bn4mVvF1Q53cnrV62_1a4x2kTXF0q3f1ublGpv7XrvIid2W18UJx_
↪xFtlKbyMwFc7HY44vee49hEwJ",
      "fulfills": {
        "output": 0,
        "txid": "04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd"
      },
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 1
  },
  "operation": "TRANSFER",
  "outputs": [
    {
      "amount": 1,
      "condition": {
```

```
        "details": {
          "bitmask": 32,
          "public_key": "3yfQPHeWAa1MxTX9Zf9176QqcpcnWcanVZZbaHb8B3h9",
          "signature": null,
          "type": "fulfillment",
          "type_id": 4
        },
        "uri": "cc:4:20:LDtSX5zaUbo0VsencVspELt-K9Bw0Y7sZLjBXcD7VCA:96"
      },
      "public_keys": [
        "3yfQPHeWAa1MxTX9Zf9176QqcpcnWcanVZZbaHb8B3h9"
      ]
    }
  ],
  "version": "0.10"
},
{
  "asset": {
    "id": "04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd"
  },
  "id": "5db2aaa9f0b0c29f5edab2a89fa46bd213890551e9e607e66e903c406a1a804b",
  "inputs": [
    {
      "fulfillment": "cf:4:LDtSX5zaUbo0VsencVspELt-
↪K9Bw0Y7sZLjBXcD7VCD586DqUyvuUOj9DiRqoAfbrS-XX_PkE-drJ9RXvtuCtxhcg16T0aCAi4_
↪WQBanUfPccanb-RgTu3D6UEgYsNAP",
      "fulfills": {
        "output": 0,
        "txid": "7eb94bc4114e221f9d9f24f5930c9bd3722b4e226ffc267e40fd3739a2a09670"
      },
      "owners_before": [
        "3yfQPHeWAa1MxTX9Zf9176QqcpcnWcanVZZbaHb8B3h9"
      ]
    }
  ],
  "metadata": {
    "sequence": 2
  },
  "operation": "TRANSFER",
  "outputs": [
    {
      "amount": 1,
      "condition": {
        "details": {
          "bitmask": 32,
          "public_key": "3Af3fhhjU6d9WecEM9Uw5hfom9kNEwE7YuDWdqAUssqm",
          "signature": null,
          "type": "fulfillment",
          "type_id": 4
        },
        "uri": "cc:4:20:IDCer8T9kVLE8s13_Jc8SbzW3Cq9Jv_MgnLYYFQXXjQ:96"
      },
      "public_keys": [
        "3Af3fhhjU6d9WecEM9Uw5hfom9kNEwE7YuDWdqAUssqm"
      ]
    }
  ],
  "version": "0.10"
```

```
}]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A list of transactions containing an asset with ID `asset_id` was found and returned.
- 400 Bad Request – The request wasn't understood by the server, e.g. the `asset_id` querystring was not included in the request.

`POST /api/v1/transactions`

Push a new transaction.

Note: The posted transaction should be structurally valid and not spending an already spent output. 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.

Example request:

```
POST /api/v1/transactions/ HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "asset": {
    "data": {
      "msg": "Hello BigchainDB!"
    }
  },
  "id": "04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd",
  "inputs": [
    {
      "fulfillment": "cf:4:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5waJ_
↪J9gSpMu3q4p4VdDO8BMroPZTaI01B2eWbvR59_yr-WZOjxVDfmrtLGSfiqsYEPdaeTS3KAMZ2Mt_
↪jv8AMH",
      "fulfills": null,
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 0
  },
  "operation": "CREATE",
  "outputs": [
    {
      "amount": 1,
      "condition": {
        "details": {
          "bitmask": 32,
          "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
          "signature": null,
```

```
          "type": "fulfillment",
          "type_id": 4
        },
        "uri": "cc:4:20:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5w:96"
      },
      "public_keys": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "version": "0.10"
}
```

Example response:

```
HTTP/1.1 202 Accepted
Content-Type: application/json

{
  "asset": {
    "data": {
      "msg": "Hello BigchainDB!"
    }
  },
  "id": "04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd",
  "inputs": [
    {
      "fulfillment": "cf:4:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5waJ_
↪J9gSpMu3q4p4VdDO8BMroPZTaI01B2eWbvR59_yr-WZOjxVDfmrtLGSfiqsYEPdaeTS3KAMZ2Mt_
↪jv8AMH",
      "fulfills": null,
      "owners_before": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
  "metadata": {
    "sequence": 0
  },
  "operation": "CREATE",
  "outputs": [
    {
      "amount": 1,
      "condition": {
        "details": {
          "bitmask": 32,
          "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
          "signature": null,
          "type": "fulfillment",
          "type_id": 4
        },
        "uri": "cc:4:20:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5w:96"
      },
      "public_keys": [
        "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
      ]
    }
  ],
```

```
  "version": "0.10"
}
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 202 Accepted – The pushed transaction was accepted in the `BACKLOG`, but the processing has not been completed.
- 400 Bad Request – The transaction was malformed and not accepted in the `BACKLOG`.

Transaction Outputs

The `/api/v1/outputs` endpoint returns transactions outputs filtered by a given public key, and optionally filtered to only include outputs that have not already been spent.

`GET /api/v1/outputs?public_key={public_key}`

Get transaction outputs by public key. The *public_key* parameter must be a base58 encoded ed25519 public key associated with transaction output ownership.

Returns a list of links to transaction outputs.

Parameters

- **`public_key`** – Base58 encoded public key associated with output ownership. This parameter is mandatory and without it the endpoint will return a `400` response code.
- **`unspent`** – Boolean value ("true" or "false") indicating if the result set should be limited to outputs that are available to spend. Defaults to "false".

Example request:

```
GET /api/v1/outputs?public_key=1AAAbbb...ccc HTTP/1.1
Host: example.com
```

Example response:

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

[
  "../transactions/
↪2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e/outputs/0",
  "../transactions/
↪2d431073e1477f3073a4693ac7ff9be5634751de1b8abaa1f4e19548ef0b4b0e/outputs/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 `public_key` querystring was not included in the request.

Statuses

GET /api/v1/statuses
Get the status of an asynchronously written transaction or block by their id.

A link to the resource is also provided in the returned payload under `_links`.

Query Parameters

- **tx_id** (*string*) – transaction ID
- **block_id** (*string*) – block ID

Note: Exactly one of the `tx_id` or `block_id` query parameters must be used together with this endpoint (see below for getting *transaction statuses* and *block statuses*).

GET /api/v1/statuses?tx_id={tx_id}

Get the status of a transaction.

The possible status values are `undecided`, `valid` or `backlog`. If a transaction in neither of those states is found, a `404 Not Found` HTTP status code is returned. We're currently looking into ways to unambigously let the user know about a transaction's status that was included in an invalid block.

Example request:

```
GET /statuses?tx_
↪id=04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd HTTP/1.1
Host: example.com
```

Example response:

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

{
  "status": "valid",
  "_links": {
    "tx": "/transactions/
↪04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd"
  }
}
```

Response Headers

- Content-Type – `application/json`
- Location – Once the transaction has been persisted, this header will link to the actual resource.

Status Codes

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

GET /api/v1/statuses?block_id={block_id}

Get the status of a block.

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

Example request:

```
GET /api/v1/statuses?block_
↪id=f027b19513833ee0a5961e4f50fe5065e8b117861bf63e2372d89bf0a0f7eed4 HTTP/1.1
Host: example.com
```

Example response:

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

{
  "status": "invalid"
}
```

Example response:

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

{
  "status": "valid",
  "_links": {
    "block": "/blocks/
↪f027b19513833ee0a5961e4f50fe5065e8b117861bf63e2372d89bf0a0f7eed4"
  }
}
```

Response Headers

- Content-Type – `application/json`
- Location – Once the block has been persisted, this header will link to the actual resource.

Status Codes

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

Advanced Usage

The following endpoints are more advanced and meant for debugging and transparency purposes.

More precisely, the *blocks endpoint* allows you to retrieve a block by `block_id` as well the list of blocks that a certain transaction with `tx_id` occured in (a transaction can occur in multiple `invalid` blocks until it either gets rejected or validated by the system). This endpoint gives the ability to drill down on the lifecycle of a transaction

The *votes endpoint* contains all the voting information for a specific block. So after retrieving the `block_id` for a given `tx_id`, one can now simply inspect the votes that happened at a specific time on that block.

Blocks

`GET /api/v1/blocks/{block_id}`
Get the block with the ID `block_id`. Any blocks, be they `VALID`, `UNDECIDED` or `INVALID` will be returned.

To check a block's status independently, use the *Statuses endpoint*. To check the votes on a block, have a look at the *votes endpoint*.

Parameters

- **block_id** (*hex string*) – block ID

Example request:

```
GET /api/v1/blocks/
↪f027b19513833ee0a5961e4f50fe5065e8b117861bf63e2372d89bf0a0f7eed4 HTTP/1.1
Host: example.com
```

Example response:

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

{
  "block": {
    "node_pubkey": "DngBurxfeNVKZWCEcDnLj1eMPAS7focUZTE5FndFGuHT",
    "timestamp": "1490777003",
    "transactions": [
      {
        "asset": {
          "data": {
            "msg": "Hello BigchainDB!"
          }
        },
        "id": "04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd",
        "inputs": [
          {
            "fulfillment": "cf:4:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5waJ_
↪J9gSpMu3q4p4VdDO8BMroPZTaI01B2eWbvR59_yr-WZOjxVDfmrtLGSfiqsYEPdaeTS3KAMZ2Mt_
↪jv8AMH",
            "fulfills": null,
            "owners_before": [
              "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
            ]
          }
        ],
        "metadata": {
          "sequence": 0
        },
        "operation": "CREATE",
        "outputs": [
          {
            "amount": 1,
            "condition": {
              "details": {
                "bitmask": 32,
                "public_key": "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD",
                "signature": null,
                "type": "fulfillment",
                "type_id": 4
              },
              "uri": "cc:4:20:MTmLrdyfhfxPw3WxnaYaQkPmU1GcEzg9mAj_O_Nuv5w:96"
            },
            "public_keys": [
              "4K9sWUMFwTgaDGPfdynrbxWqWS6sWmKbZoTjxLtVUibD"
```

```
            ]
          }
        ],
        "version": "0.10"
      }
    ],
    "voters": [
      "DngBurxfeNVKZWCEcDnLj1eMPAS7focUZTE5FndFGuHT"
    ]
  },
  "id": "f027b19513833ee0a5961e4f50fe5065e8b117861bf63e2372d89bf0a0f7eed4",
  "signature":
↪"53wxrEQDYk1dXzmvNSytbCfmNVnPqPkDQaTnAe8Jf43s6ssejPxezkCvUnGTnduNUmaLjhaan1iRLi3peu6s5DzA
↪"
}
```


Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A block with that ID was found.
- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/blocks` without the `block_id`.
- 404 Not Found – A block with that ID was not found.

GET /api/v1/blocks

The unfiltered `/blocks` endpoint without any query parameters returns a *400* status code. The list endpoint should be filtered with a `tx_id` query parameter, see the `/blocks?tx_id={tx_id}&status={UNDECIDED|VALID|INVALID}` *endpoint*.

Example request:

```
GET /api/v1/blocks HTTP/1.1
Host: example.com
```

Example response:

```
HTTP/1.1 400 Bad Request
```

Status Codes

- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/blocks` without the `block_id`.

GET /api/v1/blocks?tx_id={tx_id}&status={UNDECIDED|VALID|INVALID}

Retrieve a list of `block_id` with their corresponding status that contain a transaction with the ID `tx_id`.

Any blocks, be they `UNDECIDED`, `VALID` or `INVALID` will be returned if no status filter is provided.

Note: In case no block was found, an empty list and an HTTP status code `200 OK` is returned, as the request was still successful.

Query Parameters

- **tx_id** (*string*) – transaction ID *(required)*
- **status** (*string*) – Filter blocks by their status. One of `VALID`, `UNDECIDED` or `INVALID`.

Example request:

```
GET /api/v1/blocks?tx_
↪id=04c00267af82c161b4bf2ad4a47d1ddbfeb47eef1a14b8d51f37d6ee00ea5cdd HTTP/1.1
Host: example.com
```

Example response:

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

[
  "5d6e0cc7fc04318996c5db274d226610539e5b2ae28e561d514b552b2e488312",
  "f027b19513833ee0a5961e4f50fe5065e8b117861bf63e2372d89bf0a0f7eed4"
]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A list of blocks containing a transaction with ID `tx_id` was found and returned.
- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/blocks`, without defining `tx_id`.

Votes

GET /api/v1/votes?block_id={block_id}

Retrieve a list of votes for a certain block with ID `block_id`. To check for the validity of a vote, a user of this endpoint needs to perform the following steps:

1. Check if the vote's `node_pubkey` is allowed to vote.
2. Verify the vote's signature against the vote's body (`vote.vote`) and `node_pubkey`.

Query Parameters

- **block_id** (*string*) – The block ID to filter the votes.

Example request:

```
GET /api/v1/votes?block_
↪id=f027b19513833ee0a5961e4f50fe5065e8b117861bf63e2372d89bf0a0f7eed4 HTTP/1.1
Host: example.com
```

Example response:

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

[{
  "node_pubkey": "DngBurxfeNVKZWCEcDnLj1eMPAS7focUZTE5FndFGuHT",
```

```
  "signature":
↪"4vMbuW1h1cKinRKEfyBiwfBmag86DBpxHy6wUxYcUg7x9kmJUHhfVKjFqf126hhWea6TWqQF54zd97Tg5paQaLLV
↪",
  "vote": {
    "invalid_reason": null,
    "is_block_valid": true,
    "previous_block":
↪"0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef",
    "timestamp": "1490777003",
    "voting_for_block":
↪"f027b19513833ee0a5961e4f50fe5065e8b117861bf63e2372d89bf0a0f7eed4"
  }
}]
```

Response Headers

- Content-Type – `application/json`

Status Codes

- 200 OK – A list of votes voting for a block with ID `block_id` was found and returned.
- 400 Bad Request – The request wasn't understood by the server, e.g. just requesting `/votes`, without defining `block_id`.

Determining the API Root URL

When you start BigchainDB Server using `bigchaindb start`, an HTTP API is exposed at some address. The default is:

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

It's bound to `localhost`, so you can access it from the same machine, but it won't be directly accessible from the outside world. (The outside world could connect via a SOCKS proxy or whatnot.)

The documentation about BigchainDB Server *Configuration Settings* has a section about how to set `server.bind` so as to make the HTTP API publicly accessible.

If the API endpoint is publicly accessible, then the public API Root URL is determined as follows:

- The public IP address (like 12.34.56.78) is the public IP address of the machine exposing the HTTP API to the public internet (e.g. either the machine hosting Gunicorn or the machine running the reverse proxy such as Nginx). It's determined by AWS, Azure, Rackspace, or whoever is hosting the machine.
- The DNS hostname (like example.com) is determined by DNS records, such as an "A Record" associating example.com with 12.34.56.78
- The port (like 9984) is determined by the `server.bind` setting if Gunicorn is exposed directly to the public Internet. If a reverse proxy (like Nginx) is exposed directly to the public Internet instead, then it could expose the HTTP API on whatever port it wants to. (It should expose the HTTP API on port 9984, but it's not bound to do that by anything other than convention.)

The WebSocket Event Stream API

Important: This is currently scheduled to be implemented in BigchainDB Server 0.10.

BigchainDB provides real-time event streams over the WebSocket protocol with the Event Stream API.

Connecting to an event stream from your application enables a BigchainDB node to notify you as events are processed, such as new *validated transactions*.

Demoing the API

You may be interested in demoing the Event Stream API with the WebSocket echo test to familiarize yourself before attempting an integration.

Determining Support for the Event Stream API

In practice, it's a good idea to make sure that the node you're connecting with has advertised support for the Event Stream API. To do so, send a HTTP GET request to the node's *Root URL* and check that the response contains a `streams_<version>` property in `_links`:

```
{
  "_links": {
     "streams_v1": "ws://example.com:9984/api/v1/streams/"
  }
}
```

Connection Keep Alive

The Event Stream API initially does not provide any mechanisms for connection keep alive other than enabling TCP keepalive on each open WebSocket connection. In the future, we may add additional functionality to handle ping/pong frames or payloads designed for keep alive.

Streams

Each stream is meant as a unidirectional communication channel, where the BigchainDB node is the only party sending messages. Any messages sent to the BigchainDB node will be ignored.

Streams will always be under the WebSocket protocol (so `ws://` or `wss://`) and accessible as extensions to the `/api/v<version>/streams/` API root URL (for example, *validated transactions* would be accessible under `/api/v1/streams/valid_tx`). If you're running your own BigchainDB instance and need help determining its root URL, you can find more *here*.

All messages sent in a stream are in the JSON format.

Note: For simplicity, BigchainDB initially only provides a stream for all validated transactions. In the future, we may provide streams for other information, such as new blocks, new votes, or invalid transactions. We may also provide the ability to filter the stream for specific qualities, such as a specific `output`'s `public_key`.

If you have specific use cases that you think would fit as part of this API, feel free to reach out via gitter or email.

Valid Transactions

`/valid_tx`

Streams an event for any newly validated transactions. Message bodies contain the transaction's ID, associated asset ID, and containing block's ID.

Example message:

```
{
    "txid": "<sha3-256 hash>",
    "assetid": "<sha3-256 hash>",
    "blockid": "<sha3-256 hash>"
}
```

Note: Transactions in BigchainDB are validated in batches ("blocks") and will, therefore, be streamed in batches. Each block can contain up to a 1000 transactions, ordered by the time at which they were included in the block. The `/valid_tx` stream will send these transactions in the same order that the block stored them in, but this does **NOT** guarantee that you will recieve the events in that same order.

Community Driven Libraries and Tools

Please note that some of these projects may be work in progress, but may nevertheless be very useful.

- Javascript transaction builder
- Haskell transaction builder
- Go driver
- Java driver

CHAPTER 8

Clusters

Set Up a Cluster

This section is about how to set up a BigchainDB cluster 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.

Initial Checklist

- Do you have a governance process for making consortium-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?

Set Up the Initial Cluster

The consortium 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:

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

Backing Up & Restoring Data

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

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.

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.

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.

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 cluster nodes (owned and operated by consortium members). Only cluster nodes can produce valid signatures because only cluster 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.

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.

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.

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 consortium to decide).
- 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]

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.

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.

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.

Python Setup

The instructions that follow have been tested on Ubuntu 16.04. Similar instructions should work on similar Linux distros.

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

You must install the Python package named `fabric`, but it depends on the `cryptography` package, and that depends on some OS-level packages. On Ubuntu 16.04, you can install those OS-level packages using:

```
sudo apt-get install build-essential libssl-dev libffi-dev python-dev
```

For other operating systems, see the installation instructions for the `cryptography` package.

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).

Setting up in AWS

See the page about basic AWS Setup in the Appendices.

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.

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.

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"
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
ENABLE_WEB_ADMIN=True
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
```

If you enabled the RethinkDB web interface (by setting `ENABLE_WEB_ADMIN=True` in your AWS configuration file), then you can also check that. 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.

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).

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.)
```

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.

CHAPTER 9

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, inputs, outputs, and other things
2. Blocks
3. Votes

This section unpacks each one in turn.

The Transaction Model

A transaction has the following structure:

```
{
    "id": "<hash of transaction, excluding signatures (see explanation)>",
    "version": "<version number of the transaction model>",
    "inputs": ["<list of inputs>"],
    "outputs": ["<list of outputs>"],
    "operation": "<string>",
    "asset": "<digital asset description (explained in the next section)>",
    "metadata": "<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.
- **inputs**: List of inputs. Each *input* contains a pointer to an unspent output and a *crypto fulfillment* that satisfies the conditions of that output. A *fulfillment* is usually a signature proving the ownership of the asset. See *Inputs and Outputs*.

- **outputs**: List of outputs. Each *output* contains *crypto-conditions* that need to be fulfilled by a transfer transaction in order to transfer ownership to new owners. See *Inputs and Outputs*.
- **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**: User-provided transaction *metadata*: Can be any JSON document, or *NULL*.

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 input. 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 input 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 input.

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:

```
{
    "data": "<json document>"
}
```

For `TRANSFER` transactions we only keep the asset ID:

```
{
    "id": "<asset's CREATE transaction ID (sha3-256 hash)>"
}
```

- `id`: The ID of the `CREATE` transaction that created the asset.
- `data`: A user supplied JSON document with custom information about the asset. Defaults to null.

Inputs and Outputs

BigchainDB is modelled around *assets*, and *inputs* and *outputs* are the mechanism by which control of an asset is transferred.

Amounts of an asset are encoded in the outputs of a transaction, and each output may be spent separately. In order to spend an output, the output's `conditions` must be met by an `input` that provides corresponding `fulfillments`. Each output may be spent at most once, by a single input. Note that any asset associated with an output holding an amount greater than one is considered a divisible asset that may be split up in future transactions.

Note: This document (and various places in the BigchainDB documentation and code) talks about control of an asset in terms of *owners* and *ownership*. The language is chosen to represent the most common use cases, but in some more complex scenarios, it may not be accurate to say that the output is owned by the controllers of those public keys–it would only be correct to say that those public keys are associated with the ability to fulfill the output. Also, depending on the use case, the entity controlling an output via a private key may not be the legal owner of the asset in

the corresponding legal domain. However, since we aim to use language that is simple to understand and covers the majority of use cases, we talk in terms of *owners* of an output that have the ability to *spend* that output.

In the most basic case, an output may define a **simple signature condition**, which gives control of the output to the entity controlling a corresponding private key.

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 if and only if 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 m (of n) signatures before their asset can be transferred.

One can also put different weights on the inputs to a threshold condition, along with a threshold that the weighted-sum-of-inputs must pass for the output to be TRUE.

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).

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 complexity of the conditions, either directly by setting an allowed maximum fulfillment length, or indirectly by setting a maximum allowed transaction size which would limit the overall complexity accross all inputs and outputs of a transaction.

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.

Outputs

Note: In what follows, the list of `public_keys` (in a condition) is always the controllers of the asset at the time the transaction completed, but before the next transaction started. The list of `owners_before` (in an input) is always equal to the list of `public_keys` in that asset's previous transaction.

One New Owner

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

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

```
    "amount": "<int>"
}
```

See the reference on *outputs* for descriptions of the meaning of each field.

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:

```
{
    "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"},
        "public_keys": [
            "<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 `weight``s 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.

Inputs

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>"],
    "fulfillment":
↪"cf:4:RxFzIE679tFBk8zwEgizhmTuciAylvTUwy6EL6ehddHFJOhK5F4IjwQ1xLu2oQK9iyRCZJdfWAefZVjTt3DeG5j2exqxp
↪UrCwgnj92dnFRAEE",
    "fulfills": {
        "output": 0,
        "txid": "11b3e7d893cc5fdfcf1a1706809c7def290a3b10b0bef6525d10b024649c42d3"
    }
}
```

See the reference on *inputs* for descriptions of the meaning of each field.

Multiple Current Owners

If there are multiple *current owners*, the fulfillment will be a little different from *One Current Owner*. Suppose it has two current owners.

```
{
    "owners_before": ["<public key of the first owner before the transaction happened>
↪","<public key of the second owner before the transaction happened>"],
    "fulfillment": "cf:2:AQIBAgEBYwAEYEv6O5HjHGl7OWo2Tu5mWcWQcL_OGrFuUjyej-
↪dK3LM99TbZsRd8c9luQhU30xCH5AdNaupxg-pLHuk8DoSaDA1MHQGXUZ80a_cV-
↪4UaaaCpdey8K0CEcJxre0X96hTHCwABAWMABGBnsuHExhuSj5Mdm-
↪q0KoPgX4nAt0s00k1WTMCzuUpQIp6aStLoTSMlsvS4fmDtOSv9gubekKLuHTMAk-
↪LQFSKF1JdzwaVWAA2UOv0v_OS2gY3A-r0kRq8HtzjYdcmVswUA",
    "fulfills": {
        "output": 0,
        "txid": "e4805f1bfc999d6409b38e3a4c3b2fafad7c1280eb0d441da7083e945dd89eb8"
    }
}
```

- `owners_before`: A list of public keys of the owners before the transaction; in this case it has two owners, hence two public keys.
- `fulfillment`: A crypto-conditions URI that encodes the cryptographic fulfillments like signatures and others;'cf' indicates this is a fulfillment, '2' indicates the condition type is THRESHOLD-SHA-256 (while '4' in *One Current Owner* indicates its condition type is ED25519).
- **`fulfills`: Pointer to an output from a previous transaction that is being spent**
  - `output`: The index of the output in a previous transaction
  - `txid`: ID of the transaction

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 public keys of all nodes in the cluster>"]
  },
  "signature": "<signature of block>"
}
```

- `id`: The *hash* of the serialized inner `block` (i.e. the `timestamp`, `transactions`, `node_pubkey`, and `voters`). It's used as a unique index in the database backend (e.g. RethinkDB or MongoDB).
- **`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 all cluster nodes at the time the block was created. It's the list of nodes which can cast a vote on this block. This list can change from block to block, as nodes join and leave the cluster.
- `signature`: *Cryptographic signature* of the block by the node that created the block (i.e. the node with public key `node_pubkey`). To generate the signature, the node signs the serialized inner `block` (the same thing that was hashed to determine the `id`) using the private key corresponding to `node_pubkey`.

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.)

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 10

Transaction Schema

Transaction

A transaction represents the creation or transfer of assets in BigchainDB.

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.

Transaction.operation

type: string

Type of the transaction:

A `CREATE` transaction creates an asset in BigchainDB. This transaction has outputs but no inputs, so a dummy input is created.

A `TRANSFER` transaction transfers ownership of an asset, by providing an input that meets the conditions of an earlier transaction's outputs.

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

Transaction.asset

type: object

Description of the asset being transacted.

See: *Asset*.

Transaction.inputs

type: array (object)

Array of the inputs of a transaction.

See: *Input*.

Transaction.outputs

type: array (object)

Array of outputs provided by this transaction.

See: *Output*.

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*.

Transaction.version

type: string

BigchainDB transaction schema version.

Input

An input spends a previous output, by providing one or more fulfillments that fulfill the conditions of the previous output.

Input.owners_before

type: array (string) or null

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

Input.fulfillment

type: object or string

Fulfillment of an *Output.condition*, or, put a different way, a payload that satisfies the condition of a previous output to prove that the creator(s) of this transaction have control over the listed asset.

Input.fulfills

type: object or null

Reference to the output that is being spent.

Output

A transaction output. Describes the quantity of an asset and the requirements that must be met to spend the output.

See also: *Input*.

Output.amount

type: integer

Integral amount of the asset represented by this condition.

Output.condition

type: object

Describes the condition that needs to be met to spend the output. Has the properties:

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

Output.public_keys

type: array (string) or null

List of public keys associated with the conditions on an output.

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 contains only the user-defined payload.

Asset.id

type: string

ID of the transaction that created the asset.

Asset.data

type: object or null

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

Metadata

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

CHAPTER 11

Vote Schema

Vote

A Vote is an endorsement of a Block (identified by a hash) by a node (identified by a public key).

The outer Vote object contains the details of the vote being made as well as the signature and identifying information of the node passing the vote.

Vote.node_pubkey

type: string

Ed25519 public key identifying the voting node.

Vote.signature

type: string

Ed25519 signature of the *Vote Details* object.

Vote.vote

type: object

Vote Details to be signed.

Vote Details

Vote Details to be signed.

Vote.previous_block

type: string

ID (SHA3 hash) of the block that precedes the block being voted on. The notion of a "previous" block is subject to vote.

Vote.voting_for_block

type: string

ID (SHA3 hash) of the block being voted on.

Vote.is_block_valid

type: boolean

This field is `true` if the block was deemed valid by the node.

Vote.invalid_reason

type: string or null

Reason the block is voted invalid, or `null`.

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

Vote.timestamp

type: string

Unix timestamp that the vote was created by the node, according to the system time of the node.

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

How to Install OS-Level Dependencies

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

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

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

On Fedora 23–25, 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`.)

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 16.04, 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
```

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`.

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
```

Configuration

A one-time configuration step is required 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 \
  --interactive \
  --rm \
  --tty \
  --volume "$HOME/bigchaindb_docker:/data" \
  bigchaindb/bigchaindb \
  -y configure \
  [mongodb|rethinkdb]

Generating keypair
Configuration written to /data/.bigchaindb
Ready to go!
```

Let's analyze that command:

- `docker run` tells Docker to run some image
- `--interactive` keep STDIN open even if not attached
- `--rm` remove the container once we are done
- `--tty` allocate a pseudo-TTY
- `--volume "$HOME/bigchaindb_docker:/data"` map the host directory `$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
- `bigchaindb/bigchaindb` the image to use. All the options after the container name are passed on to the entrypoint inside the container.

- `-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
- `mongodb` or `rethinkdb` specifies the database backend to use with bigchaindb

To ensure that BigchainDB connects to the backend database bound to the virtual interface `172.17.0.1`, you must edit the BigchainDB configuration file (`~/bigchaindb_docker/.bigchaindb`) and change database.host from `localhost` to `172.17.0.1`.

Run the backend database

From v0.9 onwards, you can run either RethinkDB or MongoDB.

We use the virtual interface created by the Docker daemon to allow communication between the BigchainDB and database containers. It has an IP address of 172.17.0.1 by default.

You can also use docker host networking or bind to your primary (eth) interface, if needed.

For RethinkDB

```
docker run \
  --detach \
  --name=rethinkdb \
  --publish=172.17.0.1:28015:28015 \
  --publish=172.17.0.1:58080:8080 \
  --restart=always \
  --volume "$HOME/bigchaindb_docker:/data" \
  rethinkdb:2.3
```

You can also access the RethinkDB dashboard at http://172.17.0.1:58080/

For MongoDB

Note: MongoDB runs as user `mongodb` which had the UID `999` and GID `999` inside the container. For the volume to be mounted properly, as user `mongodb` in your host, you should have a `mongodb` user with UID and GID `999`. If you have another user on the host with UID `999`, the mapped files will be owned by this user in the host. If there is no owner with UID 999, you can create the corresponding user and group.

```
groupadd -r --gid 999 mongodb && useradd -r --uid 999 -g mongodb mongodb
```

```
docker run \
  --detach \
  --name=mongodb \
  --publish=172.17.0.1:27017:27017 \
  --restart=always \
  --volume=/tmp/mongodb_docker/db:/data/db \
  --volume=/tmp/mongodb_docker/configdb:/data/configdb \
  mongo:3.4.1 --replSet=bigchain-rs
```

Run BigchainDB

```
docker run \
  --detach \
  --name=bigchaindb \
  --publish=59984:9984 \
  --restart=always \
  --volume=$HOME/bigchaindb_docker:/data \
  bigchaindb/bigchaindb \
  start
```

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

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

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`).

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.

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)
```

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.

Hashes

BigchainDB computes transaction and block hashes using an implementation of the SHA3-256 algorithm provided by the **pysha3** package, which is a wrapper around the optimized reference implementation from http://keccak.noekeon.org.

Important: Since selecting the Keccak hashing algorithm for SHA-3 in 2012, NIST released a new version of the hash using the same algorithm but slightly different parameters. As of version 0.9, BigchainDB is using the latest version, supported by pysha3 1.0b1. See below for an example output of the hash function.

Here's the relevant code from 'bigchaindb/bigchaindb/common/crypto.py:

```
import sha3

def hash_data(data):
    """Hash the provided data using SHA3-256"""
    return sha3.sha3_256(data.encode()).hexdigest()
```

The incoming `data` is understood to be a Python 3 string, which may contain Unicode characters such as `'ü'` or `''`. The Python 3 `encode()` method converts `data` to a bytes object. `sha3.sha3_256(data.encode())` is a _sha3.SHA3 object; the `hexdigest()` method converts it to a hexadecimal string. For example:

```
>>> import sha3
>>> data = ''
>>> sha3.sha3_256(data.encode()).hexdigest()
'2b38731ba4ef72d4034bef49e87c381d1fbe75435163b391dd33249331f91fe7'
>>> data = 'hello world'
>>> sha3.sha3_256(data.encode()).hexdigest()
'644bcc7e564373040999aac89e7622f3ca71fba1d972fd94a31c3bfbf24e3938'
```

Note: Hashlocks (which are one kind of crypto-condition) may use a different hash function.

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 December 2016, EdDSA was an "Internet-Draft" with the IETF but was already widely used.

BigchainDB uses the the **cryptoconditions** package to do signature and keypair-related calculations. That package, in turn, uses the **PyNaCl** package, a Python binding to the Networking and Cryptography (NaCl) library.

All keys are represented with a Base58 encoding. The cryptoconditions package uses the **base58** package to calculate a Base58 encoding. (There's no standard for Base58 encoding.) Here's an example public/private key pair:

```
"keypair": {
    "public": "9WYFf8T65bv4S8jKU8wongKPD4AmMZAwvk1absFDbYLM",
    "private": "3x7MQpPq8AEUGEuzAxSVHjU1FhLWVQJKFNNkvHhJPGCX"
}
```

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`**(*public_key=None*, *private_key=None*, *keyring=[]*, *connection=None*, *backlog_reassign_delay=None*)

Bigchain API

Create, read, sign, write transactions to the database

__init__(*public_key=None*, *private_key=None*, *keyring=[]*, *connection=None*, *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

- **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.
- **connection** (*Connection*) – A connection to the database.

federation
Set of federation member public keys

write_transaction(*signed_transaction*)
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_new_transaction(*txid*, *exclude_block_id=None*)
Return True if the transaction does not exist in any VALID or UNDECIDED block. Return False otherwise.

Parameters

- **txid** (*str*) – Transaction ID
- **exclude_block_id** (*str*) – Exclude block from search

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

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

Returns dict if the asset exists else None.

get_spent(*txid*, *output*)
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
- **output** (*num*) – the index of the output in the respective transaction

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

get_outputs(*owner*)

Retrieve a list of links to transaction outputs for a given public key.

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

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

Return type `list` of TransactionLink

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 `output` s pointing to another transaction's condition

Return type `list` of TransactionLink

get_outputs_filtered(*owner*, *include_spent=True*)
Get a list of output links filtered on some criteria

get_transactions_filtered(*asset_id*, *operation=None*)
Get a list of transactions filtered on some criteria

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*)
Check for previous votes from this node

Parameters **block_id** (*str*) – the id of the block to check

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

Return type bool

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

Consensus

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

A consensus plugin must expose a class inheriting from this one via an entry_point.

All methods listed below must be implemented.

voting
alias of `Voting`

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

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

Pipelines

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.**tx_collector**()
A helper to deduplicate transactions

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.

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. Transaction must also not be in any VALID block.

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.**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.

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

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.

Database Backend Interfaces

Generic backend database interfaces expected by BigchainDB.

The interfaces in this module allow BigchainDB to be agnostic about its database backend. One can configure BigchainDB to use different databases as its data store by setting the `database.backend` property in the configuration or the `BIGCHAINDB_DATABASE_BACKEND` environment variable.

Generic Interfaces

bigchaindb.backend.connection

bigchaindb.backend.connection.**connect**(*backend=None*, *host=None*, *port=None*, *name=None*, *max_tries=None*, *connection_timeout=None*, *replicaset=None*, *ssl=None*, *login=None*, *password=None*)
Create a new connection to the database backend.

All arguments default to the current configuration's values if not given.

Parameters

- **backend** (*str*) – the name of the backend to use.
- **host** (*str*) – the host to connect to.
- **port** (*int*) – the port to connect to.
- **name** (*str*) – the name of the database to use.
- **replicaset** (*str*) – the name of the replica set (only relevant for MongoDB connections).

Returns An instance of *Connection* based on the given (or defaulted) `backend`.

Raises

- `ConnectionError` – If the connection to the database fails.
- `ConfigurationError` – If the given (or defaulted) `backend` is not supported or could not be loaded.

class bigchaindb.backend.connection.**Connection**(*host=None*, *port=None*, *dbname=None*, *connection_timeout=None*, *max_tries=None*, ***kwargs*)
Connection class interface.

All backend implementations should provide a connection class that from and implements this class.

__init__(*host=None*, *port=None*, *dbname=None*, *connection_timeout=None*, *max_tries=None*, ***kwargs*)

Create a new *Connection* instance.

Parameters

- **host** (*str*) – the host to connect to.
- **port** (*int*) – the port to connect to.
- **dbname** (*str*) – the name of the database to use.
- **connection_timeout** (*int, optional*) – the milliseconds to wait until timing out the database connection attempt. Defaults to 5000ms.
- **max_tries** (*int, optional*) – how many tries before giving up, if 0 then try forever. Defaults to 3.
- ****kwargs** – arbitrary keyword arguments provided by the configuration's `database` settings

run(*query*)

Run a query.

Parameters **query** – the query to run

Raises

- `DuplicateKeyError` – If the query fails because of a duplicate key constraint.
- `OperationFailure` – If the query fails for any other reason.
- `ConnectionError` – If the connection to the database fails.

connect()

Try to connect to the database.

Raises `ConnectionError` – If the connection to the database fails.

bigchaindb.backend.changefeed

Changefeed interfaces for backends.

class bigchaindb.backend.changefeed.**ChangeFeed**(*table*, *operation*, ***, *prefeed=None*, *connection=None*)

Create a new changefeed.

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.

run_forever()

Main loop of the `multipipes.Node`

This method is responsible for first feeding the prefeed to the outqueue and after that starting the changefeed and recovering from any errors that may occur in the backend.

run_changefeed()

Backend specific method to run the changefeed.

The changefeed is usually a backend cursor that is not closed when all the results are exausted. Instead it remains open waiting for new results.

This method should also filter each result based on the `operation` and put all matching results on the outqueue of `multipipes.Node`.

bigchaindb.backend.changefeed.**get_changefeed**(*connection*, *table*, *operation*, *, *prefeed=None*)

Return a ChangeFeed.

Parameters

- **connection** (*Connection*) – A connection to the database.
- **table** (*str*) – name of the table to listen to for changes.
- **operation** (*int*) – can be ChangeFeed.INSERT, ChangeFeed.DELETE, or ChangeFeed.UPDATE. Combining multiple operation is possible with the bitwise `|` operator (e.g. `ChangeFeed.INSERT | ChangeFeed.UPDATE`)
- **prefeed** (*iterable*) – whatever set of data you want to be published first.

bigchaindb.backend.query

Query interfaces for backends.

bigchaindb.backend.query.**write_transaction**(*connection*, *signed_transaction*)

Write a transaction to the backlog table.

Parameters **signed_transaction** (*dict*) – a signed transaction.

Returns The result of the operation.

bigchaindb.backend.query.**update_transaction**(*connection*, *transaction_id*, *doc*)

Update a transaction in the backlog table.

Parameters

- **transaction_id** (*str*) – the id of the transaction.
- **doc** (*dict*) – the values to update.

Returns The result of the operation.

bigchaindb.backend.query.**delete_transaction**(*connection*, **transaction_id*)

Delete a transaction from the backlog.

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

Returns The database response.

bigchaindb.backend.query.**get_stale_transactions**(*connection*, *reassign_delay*)

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.

Parameters **reassign_delay** (*int*) – threshold (in seconds) to mark a transaction stale.

Returns A cursor of transactions.

bigchaindb.backend.query.**get_transaction_from_block**(*connection*, *transaction_id*, *block_id*)

Get a transaction from a specific block.

Parameters

- **transaction_id** (*str*) – the id of the transaction.

- **block_id** (*str*) – the id of the block.

Returns The matching transaction.

bigchaindb.backend.query.**get_transaction_from_backlog**(*connection*, *transaction_id*)
Get a transaction from backlog.

Parameters **transaction_id** (*str*) – the id of the transaction.

Returns The matching transaction.

bigchaindb.backend.query.**get_blocks_status_from_transaction**(*connection*, *transaction_id*)
Retrieve block election information given a secondary index and value.

Parameters

- **value** – a value to search (e.g. transaction id string, payload hash string)
- **index** (*str*) – name of a secondary index, e.g. 'transaction_id'

Returns A list of blocks with with only election information

Return type list of dict

bigchaindb.backend.query.**get_asset_by_id**(*conneciton*, *asset_id*)
Returns the asset associated with an asset_id.

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

Returns Returns a rethinkdb cursor.

bigchaindb.backend.query.**get_spent**(*connection*, *transaction_id*, *condition_id*)
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

- **transaction_id** (*str*) – The id of the transaction.
- **condition_id** (*int*) – The index of the condition in the respective transaction.

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

bigchaindb.backend.query.**get_owned_ids**(*connection*, *owner*)
Retrieve a list of *txids* that can we used has inputs.

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

Returns A cursor for the matching transactions.

bigchaindb.backend.query.**get_votes_by_block_id**(*connection*, *block_id*)
Get all the votes casted for a specific block.

Parameters **block_id** (*str*) – the block id to use.

Returns A cursor for the matching votes.

bigchaindb.backend.query.**get_votes_by_block_id_and_voter**(*connection*, *block_id*, *node_pubkey*)
Get all the votes casted for a specific block by a specific voter.

Parameters

- **block_id** (*str*) – the block id to use.
- **node_pubkey** (*str*) – base58 encoded public key

Returns A cursor for the matching votes.

bigchaindb.backend.query.**write_block**(*connection*, *block*)
Write a block to the bigchain table.

Parameters **block** (*dict*) – the block to write.

Returns The database response.

bigchaindb.backend.query.**get_block**(*connection*, *block_id*)
Get a block from the bigchain table.

Parameters **block_id** (*str*) – block id of the block to get

Returns the block or *None*

Return type *block* (dict)

bigchaindb.backend.query.**count_blocks**(*connection*)
Count the number of blocks in the bigchain table.

Returns The number of blocks.

bigchaindb.backend.query.**count_backlog**(*connection*)
Count the number of transactions in the backlog table.

Returns The number of transactions in the backlog.

bigchaindb.backend.query.**write_vote**(*connection*, *vote*)
Write a vote to the votes table.

Parameters **vote** (*dict*) – the vote to write.

Returns The database response.

bigchaindb.backend.query.**get_genesis_block**(*connection*)
Get the genesis block.

Returns The genesis block

bigchaindb.backend.query.**get_last_voted_block**(*connection*, *node_pubkey*)
Get the last voted block for a specific node.

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

Returns The last block the node has voted on. If the node didn't cast any vote then the genesis block is returned.

bigchaindb.backend.query.**get_unvoted_blocks**(*connection*, *node_pubkey*)
Return all the blocks that have not been voted by the specified node.

Parameters **node_pubkey** (*str*) – base58 encoded public key

Returns a list of unvoted blocks

Return type list of dict

bigchaindb.backend.query.**get_txids_filtered**(*connection*, *asset_id*, *operation=None*)
Return all transactions for a particular asset id and optional operation.

Parameters

- **asset_id** (*str*) – ID of transaction that defined the asset
- **operation** (*str*) *(optional)* – Operation to filter on

bigchaindb.backend.schema

Database creation and schema-providing interfaces for backends.

bigchaindb.backend.schema.**TABLES**
tuple – The three standard tables BigchainDB relies on:

- `backlog` for incoming transactions awaiting to be put into a block.
- `bigchain` for blocks.
- `votes` to store votes for each block by each federation node.

bigchaindb.backend.schema.**create_database**(*connection*, *dbname*)
Create database to be used by BigchainDB.

Parameters **dbname** (*str*) – the name of the database to create.

Raises `DatabaseAlreadyExists` – If the given `dbname` already exists as a database.

bigchaindb.backend.schema.**create_tables**(*connection*, *dbname*)
Create the tables to be used by BigchainDB.

Parameters **dbname** (*str*) – the name of the database to create tables for.

bigchaindb.backend.schema.**create_indexes**(*connection*, *dbname*)
Create the indexes to be used by BigchainDB.

Parameters **dbname** (*str*) – the name of the database to create indexes for.

bigchaindb.backend.schema.**drop_database**(*connection*, *dbname*)
Drop the database used by BigchainDB.

Parameters **dbname** (*str*) – the name of the database to drop.

Raises `DatabaseDoesNotExist` – If the given `dbname` does not exist as a database.

bigchaindb.backend.schema.**init_database**(*connection=None*, *dbname=None*)
Initialize the configured backend for use with BigchainDB.

Creates a database with `dbname` with any required tables and supporting indexes.

Parameters

- **connection** (*Connection*) – an existing connection to use to initialize the database. Creates one if not given.
- **dbname** (*str*) – the name of the database to create. Defaults to the database name given in the BigchainDB configuration.

Raises `DatabaseAlreadyExists` – If the given `dbname` already exists as a database.

bigchaindb.backend.admin

Database configuration functions.

bigchaindb.backend.utils

exception bigchaindb.backend.utils.**ModuleDispatchRegistrationError**
Raised when there is a problem registering dispatched functions for a module

RethinkDB Backend

RethinkDB backend implementation.

Contains a RethinkDB-specific implementation of the *changefeed*, *query*, and *schema* interfaces.

You can specify BigchainDB to use RethinkDB as its database backend by either setting `database.backend` to `'rethinkdb'` in your configuration file, or setting the `BIGCHAINDB_DATABASE_BACKEND` environment variable to `'rethinkdb'`.

If configured to use RethinkDB, BigchainDB will automatically return instances of `RethinkDBConnection` for *connect()* and dispatch calls of the generic backend interfaces to the implementations in this module.

bigchaindb.backend.rethinkdb.connection

class bigchaindb.backend.rethinkdb.connection.**RethinkDBConnection**(*host=None*, *port=None*, *dbname=None*, *connection_timeout=None*, *max_tries=None*, ***kwargs*)

This class is a proxy to run queries against the database, it is:

- lazy, since it creates a connection only when needed
- resilient, because before raising exceptions it tries more times to run the query or open a connection.

run(*query*)

Run a RethinkDB query.

Parameters **query** – the RethinkDB query.

Raises `rethinkdb.ReqlDriverError` – After `max_tries`.

bigchaindb.backend.rethinkdb.schema

bigchaindb.backend.rethinkdb.query

bigchaindb.backend.rethinkdb.changefeed

class bigchaindb.backend.rethinkdb.changefeed.**RethinkDBChangeFeed**(*table*, *operation*, ***, *prefeed=None*, *connection=None*)

This class wraps a RethinkDB changefeed.

bigchaindb.backend.rethinkdb.changefeed.**get_changefeed**(*connection*, *table*, *operation*, ***, *prefeed=None*)

Return a RethinkDB changefeed.

Returns An instance of `RethinkDBChangeFeed`.

bigchaindb.backend.rethinkdb.admin

Database configuration functions.

bigchaindb.backend.rethinkdb.admin.**get_config**(*connection*, *, *table*)

Get the configuration of the given table.

Parameters

- **connection** (*Connection*) – A connection to the database.
- **table** (*str*) – The name of the table to get the configuration for.

Returns The configuration of the given table

Return type dict

bigchaindb.backend.rethinkdb.admin.**reconfigure**(*connection*, *, *table*, *shards*, *replicas*, *primary_replica_tag=None*, *dry_run=False*, *nonvoting_replica_tags=None*)

Reconfigures the given table.

Parameters

- **connection** (*Connection*) – A connection to the database.
- **table** (*str*) – The name of the table to reconfigure.
- **shards** (*int*) – The number of shards, an integer from 1-64.
- **replicas** (int | dict) –
  - If replicas is an integer, it specifies the number of replicas per shard. Specifying more replicas than there are servers will return an error.
  - If replicas is a dictionary, it specifies key-value pairs of server tags and the number of replicas to assign to those servers:

```
{'africa': 2, 'asia': 4, 'europe': 2, ...}
```

- **primary_replica_tag** (*str*) – The primary server specified by its server tag. Required if `replicas` is a dictionary. The tag must be in the `replicas` dictionary. This must not be specified if `replicas` is an integer. Defaults to `None`.
- **dry_run** (*bool*) – If `True` the generated configuration will not be applied to the table, only returned. Defaults to `False`.
- **nonvoting_replica_tags** (`list` of `str`) – Replicas with these server tags will be added to the `nonvoting_replicas` list of the resulting configuration. Defaults to `None`.

Returns

A dictionary with possibly three keys:

- `reconfigured`: the number of tables reconfigured. This will be `0` if `dry_run` is `True`.
- `config_changes`: a list of new and old table configuration values.
- `status_changes`: a list of new and old table status values.

For more information please consult RethinkDB's documentation ReQL command: reconfigure.

Return type dict

Raises `OperationError` – If the reconfiguration fails due to a RethinkDB `ReqlOpFailedError` or `ReqlQueryLogicError`.

bigchaindb.backend.rethinkdb.admin.**set_shards**(*connection*, *, *shards*, *dry_run=False*)
Sets the shards for the tables *TABLES*.

Parameters

- **connection** (*Connection*) – A connection to the database.
- **shards** (*int*) – The number of shards, an integer from 1-64.
- **dry_run** (*bool*) – If `True` the generated configuration will not be applied to the table, only returned. Defaults to `False`.

Returns

A dictionary with the configuration and status changes. For more details please see *reconfigure()*.

Return type dict

bigchaindb.backend.rethinkdb.admin.**set_replicas**(*connection*, *, *replicas*, *dry_run=False*)
Sets the replicas for the tables *TABLES*.

Parameters

- **connection** (*Connection*) – A connection to the database.
- **replicas** (*int*) – The number of replicas per shard. Specifying more replicas than there are servers will return an error.
- **dry_run** (*bool*) – If `True` the generated configuration will not be applied to the table, only returned. Defaults to `False`.

Returns

A dictionary with the configuration and status changes. For more details please see *reconfigure()*.

Return type dict

MongoDB Backend

Stay tuned!

Command Line Interface

bigchaindb.commands.bigchain

Implementation of the *bigchaindb* command, the command-line interface (CLI) for BigchainDB Server.

bigchaindb.commands.bigchain.**run_show_config**(*args*)
Show the current configuration

bigchaindb.commands.bigchain.**run_configure**(*args*, *skip_if_exists=False*)
Run a script to configure the current node.

Parameters **skip_if_exists** (*bool*) – skip the function if a config file already exists

bigchaindb.commands.bigchain.**run_export_my_pubkey**(*args*)
Export this node's public key to standard output

bigchaindb.commands.bigchain.**run_init**(*args*)
Initialize the database

bigchaindb.commands.bigchain.**run_drop**(*args*)
Drop the database

bigchaindb.commands.bigchain.**run_start**(*args*)
Start the processes to run the node

bigchaindb.commands.utils

Utility functions and basic common arguments for `argparse.ArgumentParser`.

bigchaindb.commands.utils.**configure_bigchaindb**(*command*)
Decorator to be used by command line functions, such that the configuration of bigchaindb is performed before the execution of the command.

Parameters **command** – The command to decorate.

Returns The command wrapper function.

bigchaindb.commands.utils.**start_logging_process**(*command*)
Decorator to start the logging subscriber process.

Parameters **command** – The command to decorate.

Returns The command wrapper function.

Important: Configuration, if needed, should be applied before invoking this decorator, as starting the subscriber process for logging will configure the root logger for the child process based on the state of `bigchaindb.config` at the moment this decorator is invoked.

---

bigchaindb.commands.utils.**start_rethinkdb**()
Start RethinkDB as a child process and wait for it to be available.

Raises *:class:'~bigchaindb.common.exceptions.StartupError' if* – RethinkDB cannot be started.

bigchaindb.commands.utils.**start**(*parser*, *argv*, *scope*)
Utility function to execute a subcommand.

The function will look up in the `scope` if there is a function called `run_<parser.args.command>` and will run it using `parser.args` as first positional argument.

Parameters

- **parser** – an ArgumentParser instance.
- **argv** – the list of command line arguments without the script name.
- **scope** (*dict*) – map containing (eventually) the functions to be called.

Raises `NotImplementedError` – if `scope` doesn't contain a function called `run_<parser.args.command>`.

bigchaindb.commands.utils.**mongodb_host**(*host*)
Utility function that works as a type for mongodb `host` args.

This function validates the `host` args provided by to the `add-replicas` and `remove-replicas` commands and checks if each arg is in the form "host:port"

Parameters **host** (*str*) – A string containing hostname and port (e.g. "host:port")

Raises `ArgumentTypeError` – if it fails to parse the argument

Basic AWS Setup

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

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.

Install the AWS Command-Line Interface

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

```
pip install awscli
```

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.

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.

See the page about how to generate a key pair for SSH.

Send the Public Key to AWS

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, 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.

Generate a Key Pair for SSH

This page describes how to use `ssh-keygen` to generate a public/private RSA key pair that can be used with SSH. (Note: `ssh-keygen` is found on most Linux and Unix-like operating systems; if you're using Windows, then you'll have to use another tool, such as PuTTYgen.)

By convention, SSH key pairs get stored in the `~/.ssh/` directory. Check what keys you already have there:

```
ls -1 ~/.ssh/
```

Next, make up a new key pair name (called `<name>` below). Here are some ideas:

- `aws-bdb-2`
- `tim-bdb-azure`
- `chris-bcdb-key`

Next, generate a public/private RSA key pair with that name:

```
ssh-keygen -t rsa -C "<name>" -f ~/.ssh/<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/<name>.pub` is the public key
2. `~/.ssh/<name>` is the private key

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.

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. If you're using RethinkDB, **Port 29015** can expect inbound TCP traffic from other RethinkDB nodes in the RethinkDB cluster (for RethinkDB intracluster communications).
4. If you're using MongoDB, **Port 27017** can expect inbound TCP traffic from other nodes.

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

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).

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.

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).

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.

Port 161

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

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.

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.

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.

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.

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).

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.

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.

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?

The Ubuntu ntp Packages

The Ubuntu `ntp` packages are based on the reference implementation of NTP.

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.

Example RethinkDB Storage Setups

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.

Example Using Amazon EFS

TODO

Other Examples?

TODO

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

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.

Installing BigchainDB on LXC containers using LXD

Note: This page was contributed by an external contributor and is not actively maintained. We include it in case someone is interested.

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

HTTP Routing Table

/api

Index

Symbols

B

C

D

E

F

G

H

I

M

P

R