Welcome

To get the day started, you will need to have downloaded the Vagrantbox used for the labs. If you do not have it, please flag down one of the instructors. Since conference room WiFi is always questionable, they will have the course assets on a limited number of flash drives.

Requirements:

- Vagrant 1.7.4
- VirtualBox 5.0.6 plus the VirtualBox Extensions
- Checked out copy of the `ricon_operations_lab` project

Windows Requirements:

- cygwin
- OpenSSH package

Offline Requirements:

- local copy of the custom basho/centos-6.7
  This box is based off of v2.2.2 of the bento/centos-6.7 box.
  - fetch it with `wget https://www.dropbox.com/s/rdsx5ix5bmbqql5/basho-VAGRANTSLASH-centos-6.7.box?dl=0`
  - Add it to your copy of vagrant with **`vagrant box add basho-VAGRANTSLASH-centos-6.7.box --name basho-VAGRANTSLASH-centos-6.7 --checksum-type md5 --checksum 7845698c38191c73c0f12b755fc9ef9f`**
- local copies of necessary git repositories
  - riak-inverted-index-demo
  - riak-zabbix client (included in this repository)
- local copies of CentOS 6 RPM dependencies
  - Riak 2.0.6 RHEL6 x86-64
  - Riak 2.1.1 RHEL6 x86-64

Conventions for the course material

Italic — Indicates new terms, URLs, filenames, file extensions, and occasionally, emphasis and keyword phrases

`Constant width` — Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold` — Shows commands or other text that should be typed literally by the user.

`«Constant width italic within guillemets»` — Shows text that should be replaced with user-supplied values or by values determined by context.

```
Fenced constant width — Used to indicate the result/return value of
command invocation.
```

> **Note**: This signifies a tip, suggestion, or general note.

> **Warning**: This signifies a warning or caution.

Table of contents:

- Lab 2: Configuring the cluster for the application
- Lab 3: Monitoring
- Lab 4: Sample Application
- Lab 5: Break Some Things -- well, not break, really
- Lab 6: Riak Attach is Magic
- Lab 7: Breaking Bad (destructive operations)
- Lab 8: Fixing Bad (riak*kv*vnode:repair)

Lab 0: Start up the lab environment

Locate the folder into which you copied the Vagrant environment. For the purpose of these instructions, we will assume that the environment is in a folder named *Operations_Lab* in your home directory and that your home directory can be accessed via the ~ alias.

At a shell prompt, Return the following commands:

```
cd ~/Operations_Lab
ls
```

You should have a directory listing similar to the following:

```
README.md   Vagrantfile bin         data        labs        start.sh
```

The lab environment

The lab environment consists of a Vagrantfile that will build 6 CentOS 6.7 nodes. Five to be used as Riak nodes and one to be used in the load balancer and monitoring exercises.

nodename	IP address
app	192.168.228.10
node1	192.168.228.11
node2	192.168.228.12
node3	192.168.228.13
node4	192.168.228.14
node5	192.168.228.15

To bring up the environment, type **`./start.sh`**. Vagrant will download the box file if necessary, start the virtual machines, and provision the software on the nodes. Depending on your internet connection, the first startup of the cluster will take more than 5 minutes.

The *start.sh* script just runs the *vagrant up* commands on the app node (which will download the box file if it is not present and then the nodes in parallel in order to speed up start time. If start time is less of a concern, you can simply run **`vagrant up`**

You can verify that the environment is up and running by running **`vagrant status`**. You should see the following:

```
$ vagrant status
Current machine states:

app                       running (virtualbox)
node1                     running (virtualbox)
node2                     running (virtualbox)
node3                     running (virtualbox)
node4                     running (virtualbox)
node5                     running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

Lab 1: Building a cluster

Objective: Create a five node Riak cluster from five individual nodes.

Verify that you are in the *Operations_Lab* directory, and run **`vagrant status`** to determine the state of your lab environment.

```
$ vagrant status
Current machine states:

app                       running (virtualbox)
node1                     running (virtualbox)
node2                     running (virtualbox)
node3                     running (virtualbox)
node4                     running (virtualbox)
node5                     running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

If the nodes are not all in the *running* state, run **`./start.sh`** to start all of the nodes. If all of the nodes are running, you are ready to cluster your nodes. Run **`vagrant ssh app`** to connect to the *app* node.

About tmux-cssh

To make the data entry a little more efficient, we have installed tmux-cssh on the *app* box and have created a *.tmux-cssh* file in the vagrant user's home directory. This file contains the following aliases:

```
node1:-sc 192.168.228.11
others:-sc 192.168.228.12 -sc 192.168.228.13 -sc 192.168.228.14 -sc 192.168.228.15
riak:-cs node1 -cs others
```

Let's connect to all of the nodes and check that Riak is up and running

Run:

```
tmux-cssh -u root -cs riak
```

Since it is the first time that we have connected to these nodes via SSH we will be presented with RSA key fingerprints and asked if we want to continue connecting. These sessions will have keyboard input linked to all of the panes simultaneously. Type **`yes`** and press Return.

We will now be at a root prompt on each of the nodes, as indicated by the *#* at the end. Run **`riak ping`** and press Enter. Each node should return `pong` as below:

```
[root@node1 ~]# riak ping
pong
```

If any of the nodes return something other than `pong`, please let your instructor know. If all of then nodes return `pong`, then press **`Ctrl-D`** once to exit the tmux session. You should now be back at a vagrant user prompt on app.

Since the provision script automatically installs and starts Riak on each node, we just need to connect to the **others** nodes and issue the command to join them to *node1*.

Run **`tmux-cssh -u root -cs others`** to open up a tmux session with a pane connected to all of the Riak nodes except for *node1*.

Join the nodes to *node1*

Type **`riak-admin cluster join riak@192.168.228.11`** and press Return

> **Note**: The nodename that a node identifies by is found */etc/riak/riak.conf*

Each node should reply back with a message indicating that a join request has been staged. For example, on node2 you should see the following output

```
[root@node2 ~]# riak-admin cluster join riak@192.168.228.11
Success: staged join request for 'riak@192.168.228.12' to 'riak@192.168.228.11
```

Press **`Ctrl-D`** once to exit the tmux session. You should now be back at a vagrant user prompt on app. Press **`Ctrl-D`** one more time and you should be back in the *Operations_Lab* folder on your local machine.

Plan and commit

Connect to *node1* with the **`vagrant ssh node1`** command. Once connected, open a root shell by typing **`sudo su -`** and pressing Return.

Use the **`riak-admin cluster plan`** command to output the planned cluster changes. You should get output similar to the following:

```
[root@node1 ~]# riak-admin cluster plan
=============================== Staged Changes ================================
Action         Details(s)
-------------------------------------------------------------------------------
join           'riak@192.168.228.12'
join           'riak@192.168.228.13'
join           'riak@192.168.228.14'
join           'riak@192.168.228.15'
-------------------------------------------------------------------------------


NOTE: Applying these changes will result in 1 cluster transition

###############################################################################
                         After cluster transition 1/1
###############################################################################

================================ Membership ===================================
Status     Ring    Pending    Node
-------------------------------------------------------------------------------
valid     100.0%     20.3%    'riak@192.168.228.11'
valid       0.0%     20.3%    'riak@192.168.228.12'
valid       0.0%     20.3%    'riak@192.168.228.13'
valid       0.0%     20.3%    'riak@192.168.228.14'
valid       0.0%     18.8%    'riak@192.168.228.15'
-------------------------------------------------------------------------------
Valid:5 / Leaving:0 / Exiting:0 / Joining:0 / Down:0

Transfers resulting from cluster changes: 51
  13 transfers from 'riak@192.168.228.11' to 'riak@192.168.228.13'
  13 transfers from 'riak@192.168.228.11' to 'riak@192.168.228.12'
  12 transfers from 'riak@192.168.228.11' to 'riak@192.168.228.15'
  13 transfers from 'riak@192.168.228.11' to 'riak@192.168.228.14'
```

Commit the cluster plan using **`riak-admin cluster commit`**

```
[root@node1 ~]# riak-admin cluster commit
Cluster changes committed
```

Monitor transfers with **`riak-admin transfers`**

```
[root@node1 ~]# riak-admin transfers
'riak@192.168.228.11' waiting to handoff 28 partitions

Active Transfers:
```

> **Note**: Since there is no data in these partitions, you might not actually manage to catch one in flight with the transfers command.

Once the output of *riak-admin transfers* reports that there are "No transfers active", the join operation is complete. You now have a five node riak cluster.

Press **`Ctrl-D`** twice to return to the *Operations_Lab* folder on your computer.

Lab 2: Configuring the cluster for the application

Objective: Configure our five node Riak cluster to run the Multi Backend with LevelDB as the default backend.

Since our sample application uses Secondary Indexes, we need to change the backend from the default of Bitcask.

You have two options to make the configuration changes. You can stop all of the nodes and perform all of the changes at once, or you can perform rolling restarts on the node if you need to make changes with minimal to no impact to your customer. We will perform a rolling configuration change. First, we will modify all of the configuration files and then we will perform rolling restarts of the riak process on the cluster nodes.

From the *Operations_Lab* folder:

- Connect to the *app* box with **`vagrant ssh app`**
- Start a tmux-ssh session to all of the nodes with **`tmux-cssh -u root -cs riak`**

Edit the configuration

The *riak.conf* file is a last-processed-wins configuration file format with each directive on its own line. This simplifies dynamic creation of the files because there is no complex punctuation rules to be enforced. We can just add overriding directives to the end of the file.

> **Warning**: This process will cause any data stored in the Bitcask backed to appear to become inaccessible. If you need to preserve the data in the backend that you are migrating away from, you will need to use rolling *replace* operations to stimulate transfers which will preserve the backend data. More information about this process can be found in the Riak KV documentation.

Edit the configuration file by typing **`vi /etc/riak/riak.conf`**.

- Press **`Shift-G`** to jump to the end of the file.
- Press **`$`** to jump to the end of the line.
- Press **`i`** enter Insert mode.
- Press the Right Arrow key to move over one space.
- Press Enter to move to the beginning of a new line.
- Add the following lines to the file

```
storage_backend = multi
multi_backend.default = my_leveldb
multi_backend.my_leveldb.storage_backend = leveldb
multi_backend.my_leveldb.leveldb.data_root = $(platform_data_dir)/leveldb
multi_backend.my_leveldb.leveldb.maximum_memory.percent = 50
multi_backend.my_bitcask.storage_backend = bitcask
multi_backend.my_bitcask.bitcask.data_root = $(platform_data_dir)/bitcask
```

- Press Escape to exit Insert mode,
- Type **`:wq`** and press Enter to save our changes to the file and to exit vi.
- Press **`Ctrl-D`** once to exit our tmux session and then **`Ctrl-D`** once more to exit the ssh session to *app* and return to the *Operations_Lab* folder.

Rolling Restart

Perform a rolling restart of the cluster nodes by connecting to each node in turn and restarting Riak using the following procedure. From the *Operations_Lab* folder:

- Connect to a node with **`vagrant ssh`** *«vagrant_nodename»*
- Start a root shell with **`sudo su -`**
- Stop the Riak service with **`riak stop`**
- Start the Riak service with **`riak start`**
- Wait for Riak KV to start up with **`riak-admin wait-for-service riak_kv`**
- (Optional, Recommended) Wait for transfers to complete by watching **`riak-admin transfers`**
- Press **`Ctrl-D`** two times to exit all of the running shells and return to the *Operations_Lab* folder.
- Repeat the process on the next node.

We now have our five node cluster using LevelDB as the default backend and a Bitcask backend available for use as well.

Lab 3: Monitoring

Objetive: To set up a Zabbix monitoring solution to provide our nodes.

Setting up a good monitoring solution is an enormous piece of operating a good Riak installation. This probably isn't the first time you've heard something like this, and I will be surprised if it's the last. Here. I'll make sure that won't be the last you've heard it; setting up a good monitoring solution is an enormous piece of operating a good Riak installation.

We've chosen Zabbix as our monitoring solution for this lab, primarily because it will run locally on our app nodes. It is **very** important to look closely at your needs and options before choosing a monitoring solution. Hosted platforms such as Datadog and New Relic will often serve production-level requirements much better than a locally hosted solution.

We're going to spend a lot of time on the app node, **tmux**ing into other nodes, so make sure you're ssh'd into the app node.

```
vagrant ssh app
```

Setting up the Zabbix Agents on Our Cluster

First let's setup the Zabbix Agent on all of the riak boxes. Enter a **tmux-cssh** session with all of the Riak nodes in this cluster using:

```
tmux-cssh -u root -cs riak
```

Though we already have the Zabbix Agent installed on the basho/centos-6.7 box, we still need to teach those agents how to understand the output of **riak-admin status**. The ricon_operations_lab repository includes a clone of Basho's Riak Zabbix agent repository in /vagrant/data/repos/riak-zabbix. This repository includes the template files you'll need to allow the Zabbix agent to process Riak's statistics.

> **Note**: The included clone of riak-zabbix points to the `dpb/additional_metrics` branch which, as the name suggest, provides a few more metrics and graphs than the `master` branch.

To include the Riak statistics in the set of metrics passed from the Zabbix agent to the server, all we have to do is copy *userparameter_riak.conf* from the Riak Zabbix project into the local Zabbix agent's *zabbix_agentd.d* directory.

```
cp /vagrant/data/repos/riak-zabbix/templates/userparameter_riak.conf /etc/zabbix/zabbix_agentd.d/
```

Now we need to make sure we're giving the Zabbix agent actual output to read. The agent is unable to pull directly from the /stats endpoints or from **riak-admin status**. Instead, we're going to setup an automated job that will periodically (once per minute) generate a riak-admin_status.tmp file that the Zabbix agent will extract data from.

Open up the crontab in your default editor,

```
crontab -u riak -e
```

This will open the riak user's crontab file with your default editor; probably vi. If you are unfamiliar with vi, enter Insert Mode by pressing **i**, and paste or type in the following line:

```
* * * * * /usr/sbin/riak-admin status > /var/lib/riak/riak-admin_status.new && mv /var/lib/riak/riak-admin_status.new
/var/lib/riak/riak-admin_status.tmp
```

Make sure that the cursor is at the beginning of the next (empty) line. Press **Enter** if need be.

Exit Insert Mode by pressing **Esc**

Save your changes and exit vi by typing **:wq** and then pressing **Enter**.

> **Note:** Why are we using crontab, and why so ugly? We're using crontab so the Zabbix agent doesn't have to run under escalated privileges. The hacky > then **mv** is to prevent the agent from attempting to read the tmp file while riak-admin is running, causing erroneous NULL results

Next, we're going to make a couple quick modifications to agent's config file that will allow it to connect to the Zabbix server that we're going to set up on the app node. The below Perl calls will tell Zabbix Agents to look for the server at 192.168.228.10, rather than at the local host.

```
perl -pi -e 's/Server=127.0.0.1/Server=192.168.228.10/' /etc/zabbix/zabbix_agentd.conf
perl -pi -e 's/ServerActive=127.0.0.1/ServerActive=192.168.228.10/' /etc/zabbix/zabbix_agentd.conf
```

Finally, we start the Zabbix agents.

```
service zabbix-agent start
```

Press **Ctrl+D** once to exit the tmux session.

Setting up the Zabbix Server on Our App

Zabbix servers are able to use a number of backend databases to store historical data and drive the available graphs, but it's left to the user to correctly setup said database. We'll be using the MySQL backend, because it's the one that's listed at the top of Zabbix's install instructions. I'm sorry, but I really have no further justification for this choice.

We will need to run in a privileged shell to perform the installation. Switch to a root shell using with

```
su -
```

Before starting the Zabbix server, we have to set up the MySQL database. Enter an interactive MySQL session by fist starting the MySQL daemon with,

```
service mysqld start
```

And then entering,

```
mysql
```

In that session, enter the four below commands,

```
create database zabbix character set utf8 collate utf8_bin;
grant all privileges on zabbix.* to zabbix@localhost identified by 'zabbix';
flush privileges;
exit;
```

With the database set up, we can now load it with the default set of Zabbix schemas, images, and data that will drive the Zabbix server.

```
mysql zabbix < /usr/share/doc/zabbix-server-mysql-2.4.6/create/schema.sql
mysql zabbix < /usr/share/doc/zabbix-server-mysql-2.4.6/create/images.sql
mysql zabbix < /usr/share/doc/zabbix-server-mysql-2.4.6/create/data.sql
```

With a backing database setup, we won't need to touch MySQL for the rest of this demo. We do need to tell Zabbix that database has been set up though. To do so, we just append a two configuration options to the server's configuration file,

```
echo "DBHost=localhost" >> /etc/zabbix/zabbix_server.conf
echo "DBPassword=zabbix" >> /etc/zabbix/zabbix_server.conf
```

Now we're ready to start the Zabbix server.

```
service zabbix-server start
```

With that done, we have a working Zabbix server running, but we don't have a frontend with which to control or interact with it. Luckily for us, the PHP frontend is almost completely set up for us out of the box; We just need to make one modification to the Apache's Zabbix configuration file,

> **Note**: We're setting the local timezone to Los_Angeles here because, frankly, I'm not sure PHP would know what to do with San_Francisco

```
perl -pi -e 's/# php_value date.timezone Europe\/Riga/php_value date.timezone America\/Los_Angeles/' /etc/httpd/conf.d/zabbix.conf
```

Now we get to start the HTTP daemon.

```
service httpd start
```

With all this done, we should be able to access the web frontend on our host machines through http://192.168.228.10/zabbix

With all the salient services running, we can exit the **`su -`** session by pressing **`ctrl+d`** once.

Setting Up the Web Front End

Unfortunately, this portion of the setup document is going to be somewhat more loose because the frontend in use is entirely a GUI. By nature, the descriptions provided in this document will be less precise than the above commands.

When first loading up 192.168.228.10/zabbix, you should be greeted by a Welcome page with a series of setup pages, and a `Next»` button in the bottom right of the splash screen. We're going to go ahead and page through using that button, only stopping where necessary.

1. Welcome
   Nothing needs to be done.
2. Check of pre-requisites
   Nothing needs to be done.
   Hopefully the check comes back all green. If not, something's wrong, and this guide will only be useful if you start from the beginning.
3. Configure DB connection
   This page has a few configurations options that need to be set,

   - **Database type** -- should remain **`MySQL`**
   - **Database host** -- should remain **`localhost`**
   - **Database port** -- should remain **`0`**
   - **Datbase name** -- should remain **`zabbix`**

- **User** -- needs to be changed to **`zabbix`**
- **Password** -- needs to be changed to **`zabbix`** Once all configurations have been set, press the `Test connection` button, followed by `Next»`.

4. Zabbix server details
   Nothing needs to be done.
5. Pre-Installation summary
   Nothing needs to be done.
6. Install
   Once you reach this page, the final installation is already done. We can now proceed to the Zabbix server itself.

On the next page you should be asked for a Username and Password. The default administrator credentials are going to be

Username: **`Admin`**
Password: **`zabbix`**

We're now into the dashboard. The next step it to set up a Zabbix Host to track Riak metrics and generate graphs. Before we set that host up, though, we're going to need to import the zabbix_agent_template_riak.xml file -- that came as part of the Riak Zabbix repository -- which will setup a default set of Zabbix Actors and Items to be tracked and graphed by the server. To import this template, we're going to have to enter the `Configuration->Templates` sub-tab.

> **Note**: To get there, hover your mouse over the `Configuration` tab, and a sub-menu will automatically appear immediately below. When that shows up, click on the `Templates` tab.

> **Note**: This set of metrics the zabbix_agent_template_riak.xml template traks can very easily be modified through a shell script included in the Riak Zabbix package. We'll be using the default set for now, but feel free to read up on building your own set of stats to track and graph as part of the Riak Zabbix package.

Near the upper-right of the Configuration of Templates page, under the search bar, there should be an `Import` button. Press that to open the import dialog. Press the `Choose File` button under the `Import file` form to open the file selector. Navigate to the directory this lab was downloaded to, and select **`data/repos/riak-zabbix/templates/zabbix_agent_template_riak.xml`**. With that file chosen, press the `Import` button near the bottom of the page to load the Riak template.

With the Riak template loaded, we're able to setup the Riak hosts and get tracking. To do this, we're going to have to enter the `Configuration->Hosts` sub-tab. We're going to want to create a new host with the `Create host` button that's, again, in the upper-right corner below the search bar. With this dialog open, we're going to fill in a few important fields,

- **Host name** -- **`Node 1`**
- **New Group** -- **`Riak Nodes`**
- **IP address** -- **`192.168.228.11`**

Before we add this Host, we're going to want to have it load the Riak template we imported previously. To do this, select the `Templates` tab, which will be immediately above the `Host name` form. Once there, begin typing **`Riak`** into the `Link new templates` form, and search results should begin appearing. When Riak is the only option available, press enter, and then click on the underlined `_Add_` anchor text to lock in that selection. With that done, press the `Add` button that sits below the rest of the content to create our new Riak Zabbix Host.

With that host created, we need to create 4 more; one for each other node. Because the Group and template will be remaining the same, it will be simplest to clone the Node 1 host, and modify the name and IP per new host.

Click on the newly created Node 1 host to bring up its information. Near the bottom of the pages' contents there should be a `Clone` button. Press that button, and we should be taken to a `Create New Host` page that will be seeded with the old Host's information. Update the name and IPadress according to the table below, and press `Add` to create the cloned host. Repeat this process until you have all 5 nodes added as hosts.

- **`Node 2 -- 192.168.228.12`**
- **`Node 3 -- 192.168.228.13`**
- **`Node 4 -- 192.168.228.14`**
- **`Node 5 -- 192.168.228.15`**

Congratulations! Zabbix is now up and running, and acting as a baseline monitor for our Riak cluster. It's time to explore. Check out the `Monitoring->Graphs` section, set `Group` to **`Riak Nodes`**, `Host` to any one of the running nodes, and check out what different `Graphs` are available.

Once you've gotten to know what's available, you can start putting together *Screens* that will allow you to display multiple pieces of data on one, well, screen. Because the user-interface that Zabbix has exposed for configuring these screens is tremendously slow, we've gone ahead and included a template for a useful -- if somewhat verbose -- screen. Navigate to `Configuration->Screens`, and `Import` the **`data/repos/riak-zabbix/templates/riak_large_screen_template.xml`** that's been included in the ricon_operations_lab repository, and take a look at what's been made available there.

Lab 4: The Sample Application

Objective: To install and explore both the Inverted Index sample application, and the smaller load_generator.rb application.

The Inverted Index Sample Application

The ricon_operations_lab repository includes a clone of Basho's Inverted Index demo, and should have automatically set up the application in the /home/vagrant/app/riak-inverted-index-demo directory of the app node. All required RPM packages should already be installed, as well as RVM, Ruby 1.9.3, and all required Ruby Gems.

Though the required Gems have already been installed through Bundler, we're going to need to run **`bundle install`** at least once from the application's directory to generate a Gemfile.lock file, and to verify that the correct version of the Riak Ruby Client is currently active.

> **Warning**: The basho/centos-6.7 box has two versions of the Riak Ruby client installed, one for the Inverted Index demo, and one for the load_generator.rb application. Before running either application, be sure to run **`bundle install`** from that application's root directory to verify that the correct version of the Ruby client is active.

Finalize the Inverted Index Demo Installation

If you're not already ssh'd into the *app* node, please open a new ssh session to it.

```
vagrant ssh app
```

Change directories into the Inverted Index demo folder, and verify that all Gems are correctly installed, and the Gemfile.lock has been created,

```
cd ~/app/riak-inverted-index-demo
bundle install
```

Load Some Data Into the Cluster

The Inverted Index demo expects data in a very specific format; though Riak is a schema-less database, it's very common that stored data will have at least an implicit format to follow. In this case, the format is explicitly defined in /home/vagrant/app/riak-inverted-index-demo/header.csv. There are also 10,000 records of correctly formatted data saved in the data.csv file in the same directory (and a more manually-manageable 10 stored in data-small.csv). We'll be using provided scripts to load the records in data.csv into our cluster, and the **`watch`** tool to keep track of how quickly it gets loaded.

```
ruby load_data.rb data.csv &
watch tail -n1 load_progress.txt
```

> **Note:** Now would also be a great time to check in our Zabbix graph. Pay attention to Node Gets, Node Puts, and Put FSM Times.

Experiment With the Demo Server

With the data loaded, we can now start the HTTP server that will give an interactive user-interface with which to explore this data. We'll start the server as a background process so the webpage will remain accessible as we continue to experiment on this node.

```
bundle exec unicorn -c unicorn.rb -l 0.0.0.0:8080 &
```

With that process running, you should be able to access the Inverted index application by navigating to localhost:8080 in a browser on your host machine.

From there, we can query the server either using the Secondary Indexes written directly to the Zombie objects, or using Term-Based indexing made available through the `zip_inv` bucket. Go ahead and punch in a few zip codes and see what comes up.

To get a closer look at exactly what's stored in Riak, try running the below commands from your host machine.

```
curl localhost:10018/buckets/zombies/keys/144-20-0815 -v -i

curl localhost:10018/buckets/zip_inv/keys\?keys=true

curl localhost:10018/buckets/zip_inv/keys/30083 -v -i
```

load_generator.rb

Ideally, test and QA loads generated for real-world applications structure their data to be as similar as possible to the associated production environment. For our simple demo case though, all we're after is something that will generate *some* load on out test cluster. To that end the ricon_operations_lab repository includes the reasonably simple *load_generator.rb* application.

This application will write objects of an arbitrary size to our local cluster until the process is killed with **`ctrl+c`** if it's foregrounded, or a **`kill «PID»`** command if it's backgrounded. That's really all it will do. There's no way to specify the value of the objects, but you can modify the script's behavior with the below configuration parameters (taken from running **`./load_generator.rb -h`**)

```
Usage: load_generator [options]
        --bucket_type BUCKET_TYPE    ["btype"] Name of the bucket type to target
    -b, --bucket_name BUCKET_NAME    ["load"]  Name of the bucket to target
    -s, --object_size OBJ_SIZE       [2 KiB]   Size of each PUT Object
    -n, --object_count COUNT         [500]     The count of objects to PUT before closing the current PB client, opening a new one, and repeating GETs
    -p, --puts_per_second COUNT      [200]     The *maximum* number of PUTs to perform per Second.
        --create_siblings            [false]   No coordinating GET will be performed, and siblings will be generated
```

Since the Inverted Index demo used the leveldb backend to allow for 2I queries, we ought to use this application to make PUTs against the Bitcask backend we configured in Lab 2. As can be seen, we've planned for this by setting the default value of the `--bucket-type` option to "btype" (this name is arbitrary, but in this case it's appropriately short for "Bitcask Type"). If you've been following this demo to this point we don't have a bucket type named "btype", so running the load generator as is will fail.

Creating the "btype" Bucket Type

These next few steps can be performed on any of the running Riak nodes, so be sure you're in a **`vagrant ssh app`** session and **`tmux-cssh`** to any of the nodes. We'll use Node 1 to keep things consistent.

```
vagrant ssh app
tmux-cssh -u root -cs node1
```

The next two commands will create and activate a bucket type named "btype". The `'{"props":...}'` passed into **`create`** defines the active backend to be the "my_bitcask" backend. Activating a cluster's first bucket type comes with the added caveat that the cluster will no longer be able to be downgraded to any version below the 2.0.x series. We've done our best to make sure running these commands for the fist time makes the user aware of this change.

```
riak-admin bucket-type create btype '{"props": {"backend":"my_bitcask"}}'
riak-admin bucket-type activate btype
```

No further configuration is required for the "btype" bucket type. The defined properties will very quickly be gossiped to the other nodes in the cluster.

Exit the **`tmux-cssh`** session with **`ctrl+d`**

Playing With the Load Generator

With the btype bucket type set up, we should now be able to run load_generator.rb to generate arbitrary load on the cluster and observe the effects in our Zabbix graphs. We'll need to change directories into the load_generator app and make sure its Gemfile.lock has been generated and that the correct version of the Ruby client is active.

> **Warning**: The basho/centos-6.7 box has two versions of the Riak Ruby client installed, one for the Inverted Index demo, and one for the load_generator.rb application. Before running either application, be sure to run **`bundle install`** from that application's root directory to verify that the correct version of the Ruby client is active.

```
cd ~/vagrant/app/load_generator
bundle install
```

From here, the best thing to do is start playing around. You can run the application with it's default settings,

```
./load_generator.rb
```

You can see what happens when you increase the object size to 1MB,

```
./load_generator.rb -s 1048576
```

To 5MB,

```
./load_generator.rb -s 5242880
```

To 10?

```
./load_generator.rb -s 10485760
```

You can see what happens when you fork the program to perform a lot of small PUTs into one bucket, and a few large PUTs into another (you'll have to send **`kill`** signals to the returned PIDs to stop these running),

```
./load_generator.rb -b "small" -s 2048 -l 200 &
./load_generator.rb -b "large" -s 5242880 -l 10 &
```

You can see what happens with an arbitrarily high PUT limit (hint: It might crash something. Let's find out!)

```
./load_generator.rb -b "large" -s 5242880 -l 10000000
```

Go ahead and start experimenting!

Lab 5: Break Some Things — well, not break, really

Objective: To perform a rolling upgrade of your lab environment to the latest version of Riak while the sample application is generating load.

For many applications, upgrades mean downtime. Since Riak is a clustered system, this is not necessarily so. We will use a rolling upgrade operation to show our application chugging along in spite of us performing maintenance operations.

Rolling upgrades are documented at http://docs.basho.com/riak/latest/ops/upgrading/rolling-upgrades and will feel very similar to the process that we used in Lab 2 when we changed the backend configuration.

We will start by connecting to *node1* using the **`vagrant ssh node1`** command. Once connected, we will switch to a root shell to do our maintenance by typing **`sudo su -`** and pressing **`Enter`**.

Now that we are at a root shell, we can perform the typical steps in a rolling upgrade as listed in the documentation. The following example demonstrates upgrading a Riak node that has been installed with the RHEL/CentOS packages provided by Basho.

1. Stop Riak using the **`riak stop`** command.
2. Back up the */etc/riak* and */var/lib/riak* directories with the following command:
   `sudo tar -czf riak_backup.tar.gz /var/lib/riak /etc/riak`
3. Use RPM or Yum to Upgrade Riak
   `sudo rpm -Uvh /vagrant/data/rpmcache/riak-2.1.1-1.el6.x86_64.rpm`
4. Restart Riak with the **`riak start`** command.
5. Verify that Riak is running the new version with **`riak version`**
6. Wait for the *riak_kv* service to start. There is a helper command that will poll the status of a Riak service and notify you when it is started:
   `riak-admin wait-for-service riak_kv`
7. While the node was offline, other nodes may have accepted writes on its behalf. This data is transferred to the node when it becomes available. Wait for any hinted handoff transfers to complete. You can monitor them with the **`riak-admin transfers`** command.

Press **`Ctrl-D`** twice—once to exit the root shell, and again to return to the *Operations_Lab* folder.

Repeat the process for the remaining nodes (*node2*, *node3*, *node4*, and *node5*) in the cluster.

Once you have completed this process you will now have your five-node cluster running on the latest version of Riak.

Lab 6: Riak Attach is Magic

The *riak attach* command will allow you to connect to the running Riak's Erlang VM and issue commands. It is both the most amazing feature and most unnerving feature at the same time. However, with a little bit of knowledge, you will be able to work well with the Basho support team and be able to understand arcane snippets such as:

```
{ok,Ring} = riak_core_ring_manager:get_my_ring(),
Locals = [{Idx,Pid,riak_core_ring:index_owner(Ring,Idx)} || {Idx,Pid} <-
  riak_core_vnode_manager:all_index_pid(riak_kv_vnode)],
rp([ {Idx,Pid,Owner} || {Idx,Pid,Owner} <- Locals, Owner =/= node()]).
```

To that end, we should probably talk a little bit about Erlang.

Erlang Primer

Data Types

Terms

A piece of data of any data type is called a term.

Number

There are two types of numeric literals, integers and floats. Besides the conventional notation, there are two Erlang-specific notations:

$char - ASCII value or unicode code-point of the character char.
base#value - Integer with the base base, that must be an integer in the range 2..36.

Examples:

```
1> 42.
42
2> $A.
65
3> $\n.
10
4> 2#101.
5
5> 16#1f.
31
6> 2.3.
2.3
7> 2.3e3.
2.3e3
8> 2.3e-3.
0.0023
```

Atom

An atom is a literal, a constant with name. An atom is to be enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (_), or @.

Examples:

```
hello
phone_number
'Monday'
'phone number'
'riak@192.168.228.11'
```

Bit Strings and Binaries

A bit string is used to store an area of untyped memory. Bit strings are expressed using the bit syntax. Bit strings that consist of a number of bits that are evenly divisible by eight, are called binaries

Examples:

```
1> <<10,20>>.
<<10,20>>
2> <<"ABC">>.
<<"ABC">>
1> <<1:1,0:1>>.
<<2:2>>
```

Pid

A process identifier, pid, identifies a process. They are generally displayed as three dot-seperated numeric values inside of angle brackets. For example: <0.101.2>

Tuple

A tuple is a compound data type with a fixed number of terms:

{Term1,...,TermN}

Each term Term in the tuple is called an element. The number of elements is said to be the size of the tuple. There exists a number of BIFs (built-in functions) to manipulate tuples.

Examples:

```
1> P = {adam,24,{july,29}}.
{adam,24,{july,29}}
2> element(1,P).
adam
3> element(3,P).
{july,29}
4> P2 = setelement(2,P,25).
{adam,25,{july,29}}
5> tuple_size(P).
3
6> tuple_size({}).
0
```

List

A list is a compound data type with a variable number of terms.

[Term1,...,TermN] Each term Term in the list is called an element. The number of elements is said to be the length of the list.

Formally, a list is either the empty list [] or consists of a head (first element) and a tail (remainder of the list). The tail is also a list. The latter can be expressed as [HIT]. The notation [Term1,...,TermN] above is equivalent with the list [Term1|[...|[TermN|[]]]].

Example:

```
[] is a list, thus
[c|[]] is a list, thus
[b|[c|[]]] is a list, thus
[a|[b|[c|[]]]] is a list, or in short [a,b,c]
```

A list where the tail is a list is sometimes called a proper list. It is allowed to have a list where the tail is not a list, for example, [alb]. However, this type of list is of little practical use.

Examples:

```
1> L1 = [a,2,{c,4}]. [a,2,{c,4}] 2> [H|T] = L1. [a,2,{c,4}] 3> H. a 4> T. [2,{c,4}] 5> L2 = [d|T]. [d,2,{c,4}] 6> length(L1). 3 7> length([]
```

String

Strings are enclosed in double quotes ("), but is not a data type in Erlang. Instead, a string "hello" is shorthand for the list [$h,$e,$l,$l,$o], that is, [104,101,108,108,111].

Two adjacent string literals are concatenated into one. This is done in the compilation, thus, does not incur any runtime overhead.

Example:

```
"string" "42"
is equivalent to

"string42"
```

Record

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a struct in C. However, a record is not a true data type. Instead, record expressions are translated to tuple expressions during compilation. Therefore, record expressions are not understood by the shell unless special actions are taken. For details, see the shell(3) manual page in STDLIB).

Boolean

There is no Boolean data type in Erlang. Instead the atoms true and false are used to denote Boolean values.

Escape Sequences

Within strings and quoted atoms, the following escape sequences are recognized:

Sequence	Description
\b	Backspace
\d	Delete
\e	Escape
\f	Form feed
\n	Newline
\r	Carriage return
\s	Space
\t	Tab
\v	Vertical tab
\XYZ, \YZ, \Z	Character with octal representation XYZ, YZ or Z
\xXY	Character with hexadecimal representation XY
\x{X...}	Character with hexadecimal representation; X... is one or more hexadecimal characters
^a...^z	Control A to control Z
^A...^Z	Control A to control Z
\'	Single quote
"	Double quote
\	Backslash

Table 3.1: Recognized Escape Sequences

Type Conversions

There are a number of BIFs for type conversions.

Examples:

```
1> atom_to_list(hello).
"hello"
2> list_to_atom("hello").
hello
3> binary_to_list(<<"hello">>).
"hello"
4> binary_to_list(<<104,101,108,108,111>>).
"hello"
5> list_to_binary("hello").
<<104,101,108,108,111>>
6> float_to_list(7.0).
"7.00000000000000000000e+00"
7> list_to_float("7.000e+00").
7.0
8> integer_to_list(77).
"77"
9> list_to_integer("77").
77
10> tuple_to_list({a,b,c}).
[a,b,c]
11> list_to_tuple([a,b,c]).
{a,b,c}
12> term_to_binary({a,b,c}).
<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>
13> binary_to_term(<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>).
{a,b,c}
14> binary_to_integer(<<"77">>).
77
15> integer_to_binary(77).
<<"77">>
16> float_to_binary(7.0).
<<"7.00000000000000000000e+00">>
17> binary_to_float(<<"7.000e+00>>").
7.0
```

We will use some of this information while we damage and fix our cluster.

A hitchhiker's guide to riak attach

The *riak attach* command will connect you to a REPL (read-evaluate-print-loop) running inside of the Erlang VM that is running the Riak application.

Connect to *node1* using the **`vagrant ssh node1`** command. Type **`sudo su -`** and press **`Enter`** to start a root shell.

Run the **`riak attach`** command which will connect you to the Erlang VM. Once commected you should see a prompt similar to the following:

```
Remote Shell: Use "Ctrl-C a" to quit. q() or init:stop() will terminate the riak node.
Erlang R16B02_basho8 (erts-5.10.3) [source] [64-bit] [async-threads:10] [kernel-poll:false] [frame-pointer]

Eshell V5.10.3  (abort with ^G)
(riak@192.168.228.11)1>
```

So first things first. Let's learn how to get out of the shell. Press **`Ctrl-G`** and then **`q`** then press Enter to exit *riak attach*. Try that now.

You should be back at the root user prompt on *node1*. Restart a riak attach session by running **`riak attach`**

So what can we do in here.

This is a full-featured Erlang environment, so you can run simple Erlang commands. Type the following:

```
100 * 200.
```

Note that the period (also referred to as a full-stop) is significant to the language.

Press Enter and you should see the following:

```
(riak@192.168.228.11)1> 100 * 200.
20000
(riak@192.168.228.11)2>
```

A typical use of riak attach is to run snippets provided by Basho support. One such snippet will collect the partitions owned by a specific node. We will use this snippet in a later lab:

```
Ring = riak_core_ring_manager:get_my_ring().
Partitions = [P || {P, 'riak@192.168.211.11'} <- riak_core_ring:all_owners(Ring)].
[riak_kv_vnode:repair(P) || P <- Partitions].
```

There are also interesting abilities to perform commands using RPC since the nodes are all clustered. We are going to leave further discussion about the possibilities of riak-attach for the Q&A

Lab 7: Breaking Bad...

Objective: To perform some destructive changes to the cluster. We will join a node in place and join a node with a new address.

Sometimes, in spite of our best efforts, a node just fails -- fails spectacularly. In this case, Riak can easily survive single node outages with default configurations.

It is worth mentioning the caveats that we saw in the rolling upgrade exercise. While there is replica loss, coverage based queries will report inconsistent results.

In order to simulate a "spectacular" hardware failure, we are going to delete one of the nodes from the lab environment. From the *Operations_Lab* folder, run the following commands:

```
vagrant destroy node3 -f
```

If we check in with the application, we should be doing GETs and PUTs just fine. Coverage queries are going to become inconsistent because of the replica loss we just put the cluster through.

Now let's get even more exciting... let's cause a little more loss. Destroy a second node using this command:

```
vagrant destroy node4 -f
```

Now we should start seeing some soft failures. However, as we watch, the number of soft failures will decrease. This is due to read-repair repairing data into fallback partitions on the remaining nodes. When the downed nodes come back, this data will be handed back off to them.

Rejoin Node3 to the Cluster

Let's reprovision node3 so that it can be joined back into the cluster. Run:

```
vagrant up node3
```

This will provision the node and get us ready to rejoin the node to the cluster. Once the node is provisioned, run

```
vagrant ssh node3
sudo su -
riak-admin cluster join riak@192.168.228.11
riak-admin cluster plan
```

You will receive an error:

```
[root@node3 ~]# riak-admin cluster plan
Cannot plan until cluster state has converged.
Check 'Ring Ready' in 'riak-admin ring_status'
```

Since there is another node down, the rind will never become "ready" until we indicate that that node is not expected to come back.

To mark node4 as down, run

```
riak-admin cluster down riak@192.168.228.14
```

Marking a node down is only necessary when we need to perform a ring transition (generally to effect a membership change) while a cluster member is down. Let's verify that the ring is

now ready by running the `riak-admin ring-status` command. We should now receive output that indicated that ring-ready is *true*.

```
[root@node3 ~]# riak-admin ring-status
================================== Claimant ===================================
Claimant:  'riak@192.168.228.11'
Status:     up
Ring Ready: true

============================== Ownership Handoff ==============================
No pending changes.

============================== Unreachable Nodes ==============================
All nodes are up and reachable
```

Now, try planning and committing the cluster changes:

```
riak-admin cluster plan
riak-admin cluster commit
```

Once that is done, the node will rejoin the cluster and participate fully. It will also hand off any data back into the cluster that it received while it was a cluster of one. Monitor the output of `riak-admin transfers` to determine when handoff is complete.

Renaming Nodes During Replacement

Now... Suppose that we were not able to reprovision node4 in place with the same IP address. Let's change the Vagrantfile in the *Operations_lab* folder to bring node4 back up with a different IP address.

Using your text editor of choice, change the following line from:

```
    node4.vm.network "private_network", ip: "192.168.228.14"
```

to

```
    node4.vm.network "private_network", ip: "192.168.228.16"
```

Save the file and edit the text editor. Now, run `vagrant up node4` This will cause the node to be reprovisioned as before, but using the new IP address. The provisioning scripts are smart enough to configure the nodename to utilize the new IP address and Riak starts automatically as before.

If we join this node to the cluster, it will come in as a brand new node and have some measure of ownership assigned to it. What we'd rather have happen is for all of the partitions formerly assigned to *node4* to simply be assigned to this new incarnation of *node4*.

To have riak do what we'd like, we are going to use a force-replace operation. From the *Operations_Lab* folder, run the following command:

```
vagrant ssh node4
```

Now that you are connected to *node4*, run this series of commands:

```
sudo su -
riak-admin cluster join riak@192.168.228.11
```

> **Warning**: Do not commit the plat at this point.

Run the *force-replace* of the old *Node4* node name with the new *Node4* node name.

```
riak-admin cluster force-replace riak@192.168.228.14 riak@192.168.228.16
```

Generate and display the planned changes to the cluster.

```
riak-admin cluster plan
```

The cluster plan will show that you are replacing the old instance of *node4* with the new instance.

```
============================== Staged Changes ==============================
Action         Details(s)
-------------------------------------------------------------------------------
force-replace  'riak@192.168.228.14' with 'riak@192.168.228.16'
join           'riak@192.168.228.16'
-------------------------------------------------------------------------------

WARNING: All of 'riak@192.168.228.14' replicas will be lost

NOTE: Applying these changes will result in 1 cluster transition

###############################################################################
                         After cluster transition 1/1
###############################################################################

================================ Membership ================================
Status     Ring    Pending    Node
-------------------------------------------------------------------------------
valid      20.3%      --      'riak@192.168.228.11'
valid      20.3%      --      'riak@192.168.228.12'
valid      20.3%      --      'riak@192.168.228.13'
valid      18.8%      --      'riak@192.168.228.15'
valid      20.3%      --      'riak@192.168.228.16'
-------------------------------------------------------------------------------
Valid:5 / Leaving:0 / Exiting:0 / Joining:0 / Down:0
```

The cluster plan will indicate that you are force replacing the node and will warn you that any replicas on

Commit the plan.

```
riak-admin cluster commit
```

Once that is done, the node will have all of the ownership associated with the dead node's nodename aggigned to it. It will also hand off any data back into the cluster that it received while it was a cluster of one. Monitor the output of `riak-admin transfers` to determine when handoff is complete.

Unfortunately, we are still going to have inconsistency unless we read every object in our cluster. Well, we actually have a plan for that that we discuss in our final lab of the day.

At this point, you should have some significant replica loss in your cluster and be experiencing inconsistent key listing returns.

Lab 8: Fixing Bad (riak_kv_vnode:repair)

As part of the previous **Breaking Bad** lab, we caused some damage to our clusters' replicas. This, unfortunately, does sometimes happen to production clusters (hopefully not intentionally). As you have learned, Riak was designed to withstand -- or at the very least reduce the severity of -- many types of system failures, and understanding how Riak is capable of recovering from these situations can be a valuable piece of knowledge.

Riak has two primary anti-entropy mechanisms; passive *Read Repair* and *Active Anti-Entropy*.

Read Repair occurs as part of normal GET and PUT operations in Riak, using the causal history that Riak objects carry. If two or more replicas of an object are found to be in a resolvable conflict (e.g. one of the replicas missed a PUT and is carrying a stale version of the object, and one causal history *dominates* the other), the object will be repaired on the offending node as part of that operation. This form of anti-entropy is call *passive* Read Repair because no Riak systems drive this process; it's entirely dependent on users performing GETs or PUTs on objects with divergent replicas.

Active Anti-Entropy (AAE for short) was designed to mitigate the shortcomings of passive Read Repair. For example, in usecases that store data for long periods of time but rarely touch that data, it could takes months for a GET to be performed on a replica that fell out of date or a that was lost. The AAE subsystem actively looks for replicas that disagree across data partitions by building Hash Trees (Merkle Trees specifically, hashing an object's identifier, causal history, and value) of every partition, and performing comparisons across objects' preflists. When a divergent replica is found, the same mechanisms that are used in passive Read Repair are used by the AAE subsystem to repair that replica.

Sometimes, these system aren't enough. If your cluster has experienced widespread loss of replicas or the loss of an entire partition, you may need to perform a *Repair Operation* on one or more of your data partitions.

Running a Repair

The Riak KV repair operation will repair objects from a node's adjacent partitions on the ring, rebuilding the data that should be in the damaged partition. This is done as efficiently as possible by generating a hash range for all the buckets, and thus avoiding a preflist calculation for each key. Only a hash of each key is done, its range determined from a bucket->range map, and then the hash is checked against the range.

To perform a Repair Operation, you'll need to know the ID of the partition that has been damaged. In our case, the damaged partitions are all of the partitions contained on nodes 3 and

4. We'll be using a **`riak attach`** session to extract the exact partitions.

If you don't currently have a connection, connect to node1 with the **`vagrant ssh node1`** command.

Once that connection is made, open a Riak Attach session with the **`riak attach`** command.

From that Riak Attach session, we'll first find out which partitions need to be repaired. Run the below list comprehensions to generate the full list,

```
{ok, Ring} = riak_core_ring_manager:get_my_ring().
Node3Partitions = [P || {P, 'riak@192.168.228.13'} <- riak_core_ring:all_owners(Ring)].
Node4Partitions = [P || {P, 'riak@192.168.228.14'} <- riak_core_ring:all_owners(Ring)].
AllPartitions = Node3Partitions ++ Node4Partitions.
riak_kv_vnode:repair(P) || P <- AllPartitions].
```

Once the command has been executed, detach from the Riak Attach session by pressing Control-G, q, and then Enter.

Killing a Repair

Repair Operations are often discouraged because they can be very resource intensive. Currently there is no easy way to kill an individual repair; the only option is to kill all repairs targeting a given node.

This is done by opening a Riak Attach session on the node performing the repair, and running **`riak_core_vnode_manager:kill_repairs(killed_by_user).`**. I