CS300 Couchbase NoSQL Server Administration

Lab 7 Exercise Manual

Release: 6.5.1

Revised: June 22nd, 2020

Lab #7: Advanced XDCR and Backup/Restore

Objective: This 1 hour lab will walk you through a couple of more advanced XDCR scenarios and then cover how to backup a bucket's files and restore it to a live cluster.

Overview: **The following high-level steps are involved in this lab:**

- Study optimistic replication in further detail
- Use pillowfight to insert 100,000 items UNDER the optimistic replication threshold into the 6-node NYC cluster so that the metadata is NOT checked on the NYC side before optimistically replicating it to the destination cluster
- Use pillowfight to insert 100,000 items OVER the optimistic replication threshold into the 6-node NYC cluster so that the metadata is checked on the NYC side before replicating it to the destination cluster
- Demonstrate that XDCR replication keeps occurring to remaining nodes even after a node goes down and fails over
- Learn how to use the vbucketkeygen tool to insert 1 key into each of the 1024 vBuckets
- Learn about cbbackup and cbrestore
- Use cbbackup to backup data in a bucket to some files
- Use cbrestore to restore the files back into a live bucket

Studying Optimistic Replication:

Remember that the Optimistic Replication threshold in our bidirectional replication stream is set to the default of 256 bytes. This parameter is basically used to tune the tradeoff between latency and bandwidth. If you want the fastest latency, you will have to sacrifice some bandwidth. On the other hand, if you want to conserve bandwidth, you will have to suffer some latency.

If the item that you write/update to either side cluster (NYC or London) is under 256 bytes, then the item is optimistically just thrown into the destination cluster's replication queue. However if the item you write/update is larger than 256 bytes, then the source cluster will first do a read of the same key's metadata over the WAN (Wide Area Network) to check if the destination cluster would win the conflict resolution. If the destination cluster's revision count is lower than the source cluster (where you did the write/update), then the source will send the item all the way down the WAN to the destination cluster. Before the destination cluster pushes that write/update to disk, it will do another metadata read locally to make sure that while the item was transiting over the WAN it didn't get updated with a higher revision count locally. If the destination cluster decides that the item it got from the source is still the higher revision count, then the destination cluster pushes that item into its local disk write queue.

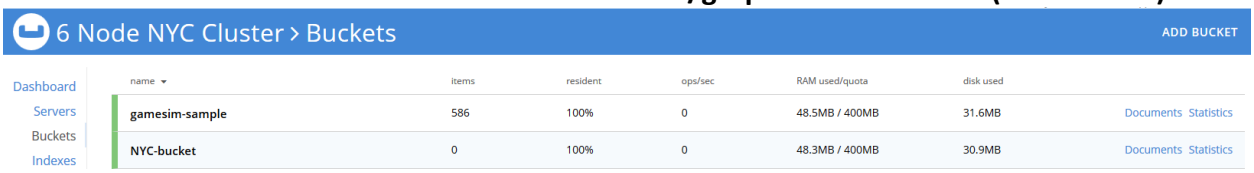
At this point, NYC-bucket and London-bucket should both be empty with zero items in either bucket.

Let's use pillowfight from the App Server to first send 100,000 keys of size 200 bytes (under the optimistic replication threshold) to the 6-node NYC cluster. In this case, the NYC cluster will NOT attempt to query metadata for any of the items from the remote side. It will just optimistically push all 100,000 items over to the London side and let London's cluster deal with it.

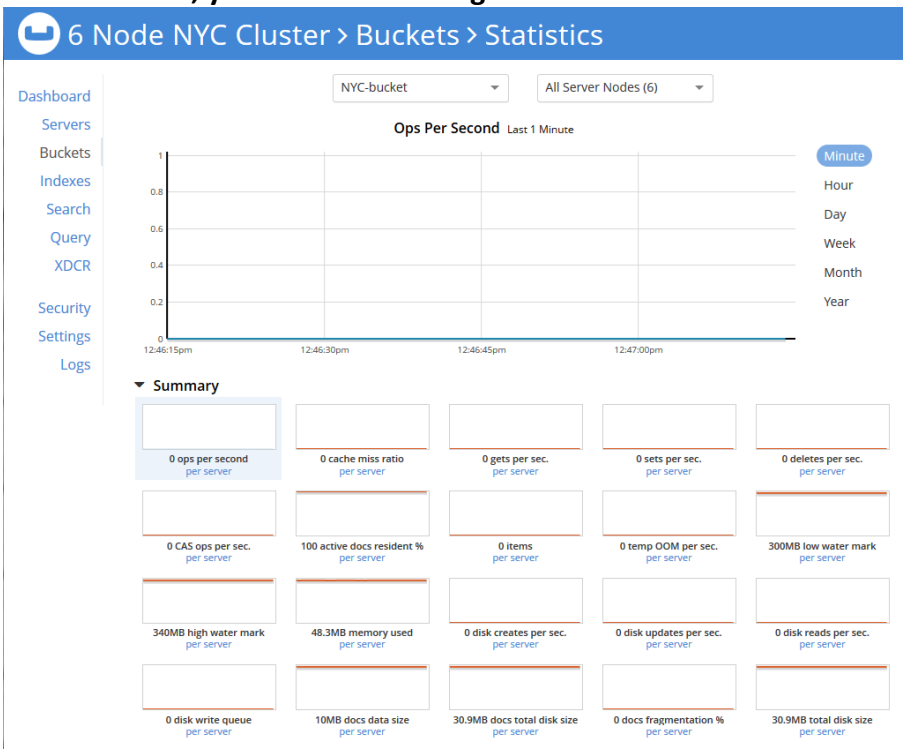
Later in the lab we will also send keys of size over 256 bytes to see how that has a different effect in the cluster.

Before we generate any data, let's set up our Web UI properly to see the effects of optimistic replication.

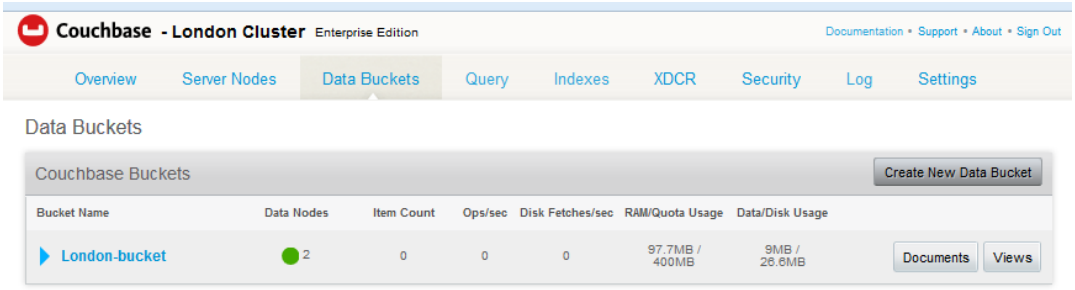
Switch browser tabs to the 6-node NYC cluster's Web UI, click on "Buckets link" at the side menu and click on the "statistics" to load the charts/graphs for this bucket(NYC-bucket):



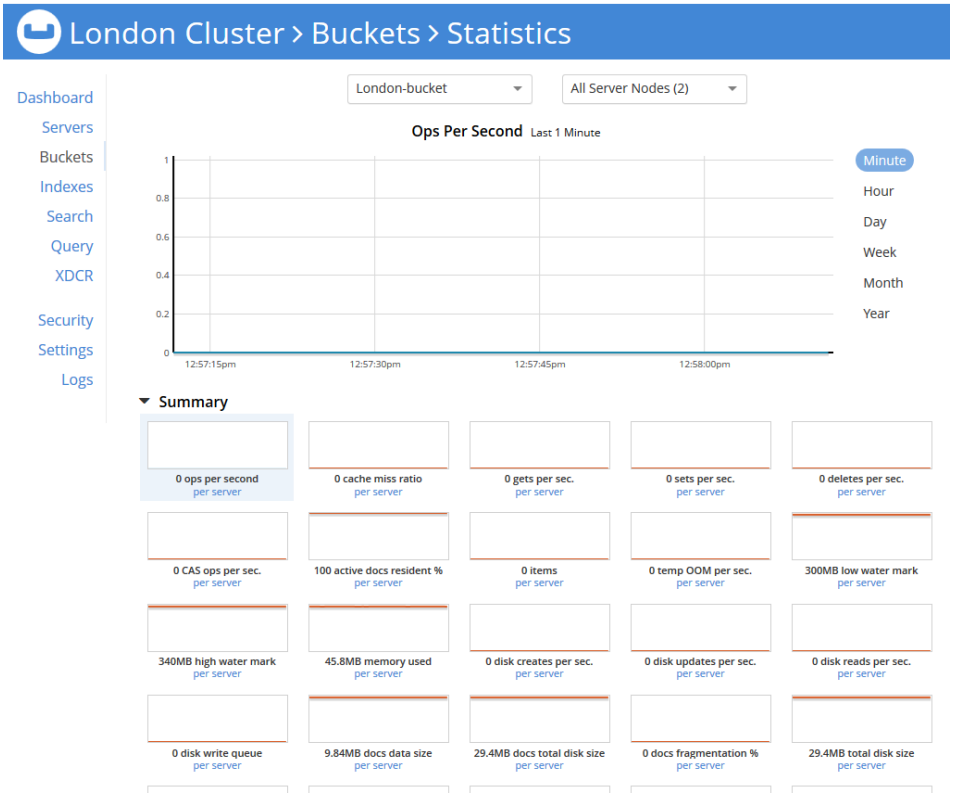
In the first tab, you should be seeing metrics for the "NYC-bucket" for "All Server Nodes":



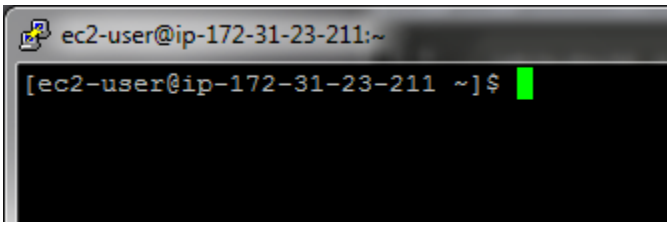
Next, switch to the 2nd browser tab for the London cluster's Web UI, click on "Buckets" at the side menu and click on the "statistics" to load the charts/graphs for the London-bucket:



In the second tab, you should be seeing metrics for the "London-bucket" for "All Server Nodes":



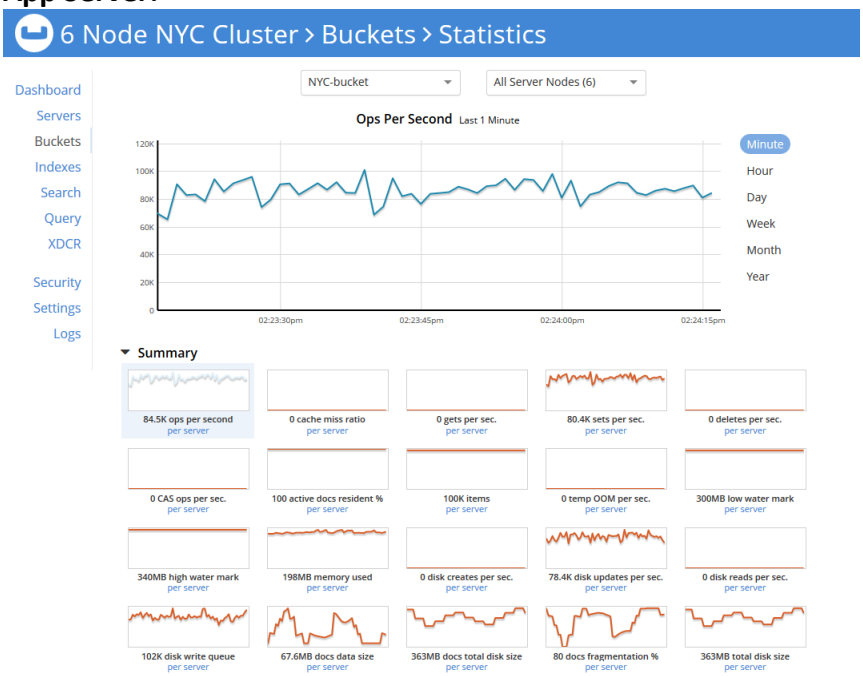
Switch to the App Server (black VM):



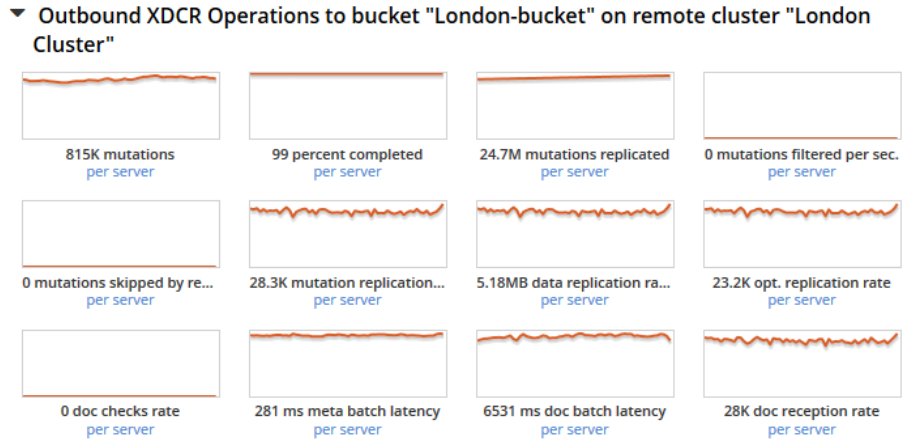
Run pillowfight to write 100,000 items, in 10 iteration, with a 100% write (set) ratio and a maximum payload size of 200 bytes. There will be a key prefix of "from_NYC" in front of all the keys. Use the public hostname of the 1st node in the 6-node NYC cluster in the command. So, we will push the writes into the 6-node NYC cluster and expect them to be optimistically replicated into the 2-node London cluster without first checking the metadata.

```
[ec2-user@appserver ~]$ cbc pillowfight -U couchbase://$NODE1/NYC-bucket --num-items=100000 --batch-size=10000 --set-pct=100 --min-size=50 --max-size=200 --timings --num-cycles=10 -p from_NYC -l
```

While the above command is generating data, quickly switch to the 6-node NYC cluster's Web UI and observe that the cluster is receiving 40-70K writes per second across 4 nodes from the App Server:

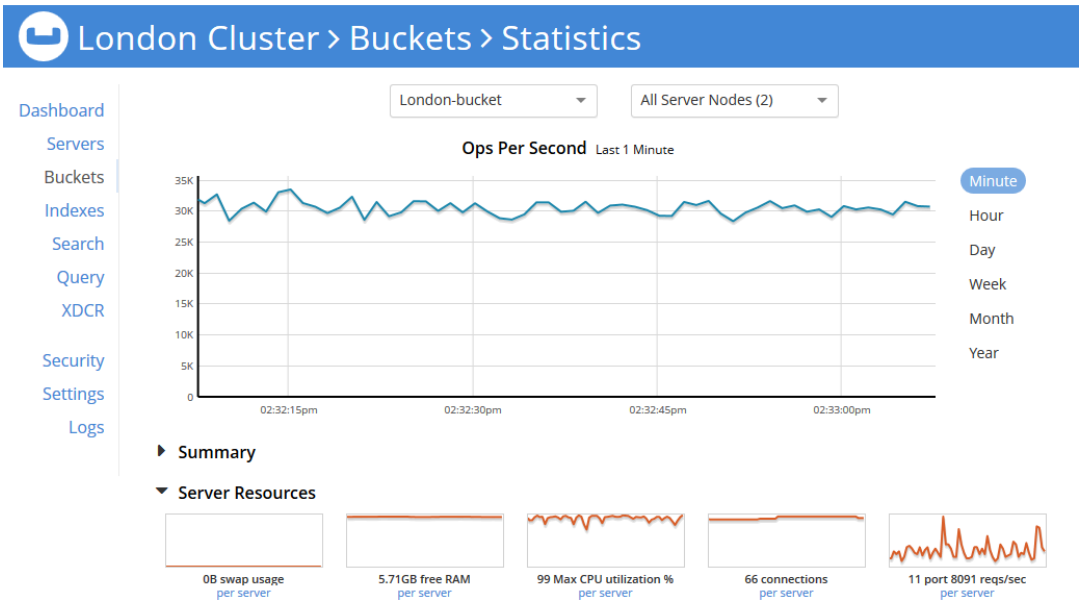


Then, in the same 6-node cluster's Web UI, scroll down and click the blue arrow to expand "OUTBOUND XDCR OPERATIONS" to the London-bucket:



Observe in the screenshot above that the graph in the 2nd row, 4th column shows mutations replicated optimistically so far.
The graph in the 3rd row, 2nd column shows 250-300 millisecond latency for metadata operations.

Next, switch to the 2nd browser tab for the 2-node London cluster and in the large "ops per second graph" observe that it is receiving around 30K operations (writes) per second from the NYC side:

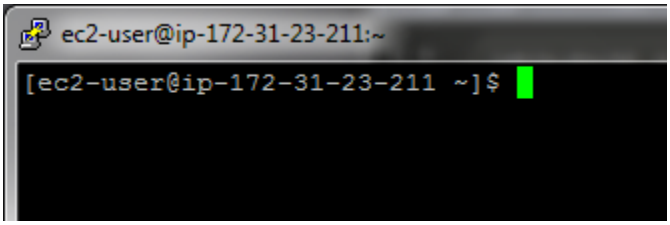


Scroll down on the same page, and expand "INCOMING XDCR OPERATIONS" and notice that the London cluster is getting around 25-30K sets per second (at least in my specific cluster at the time I captured the screenshot):

The "metadata reads per sec" graph in the 1st column above is NOT referring to metadata reads occurring on the destination/London side, but rather metadata reads happening from source/NYC.

Next we'll do a similar test of writing 100,000 keys into the NYC cluster with pillowfight, but this time we'll set the item size to 500 bytes, well above the optimistic replication threshold. And actually we won't write 100,000 new keys, but rather update the 100,000 keys from before (so we'll use the same key prefix). This time the NYC cluster will have to keep doing remote metadata reads a total of 100,000 times before sending each item to the destination (London) cluster.

Switch to the App Server (black VM):

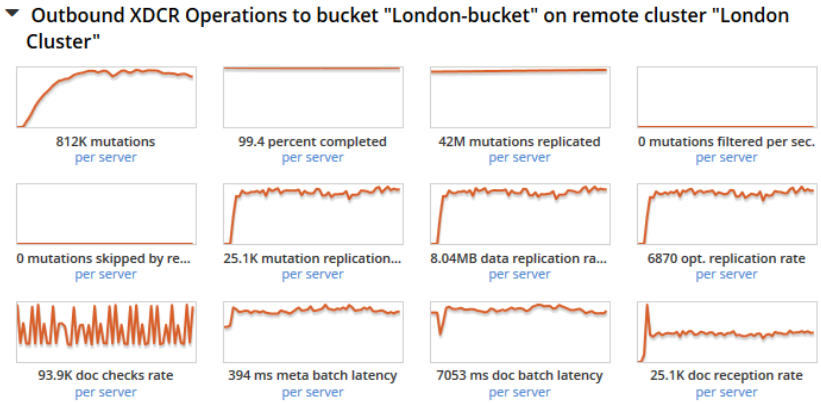


Run pillowfight to update the 100,000 items, in 10 iterations, with a 100% write (set) ratio and a maximum payload size of 10,000 bytes. There will be a similar key prefix of "from_NYC" in front of all the keys. Use the public hostname of the 1st node in the 6-node NYC cluster in the command. So, we will push the writes into the 6-node NYC cluster and expect 100,000 metadata reads to happen remotely from the 2-node London cluster before replicating the items over to the 2-node London cluster.

```
[ec2-user@appserver ~]$ cbc pillowfight -U couchbase://$NODE1/NYC-bucket --num-items=100000 --batch-size=10000 --set-pct=100 --min-size=50 --max-size=4000 --timings --num-cycles=10 -p from_NYC -l
```

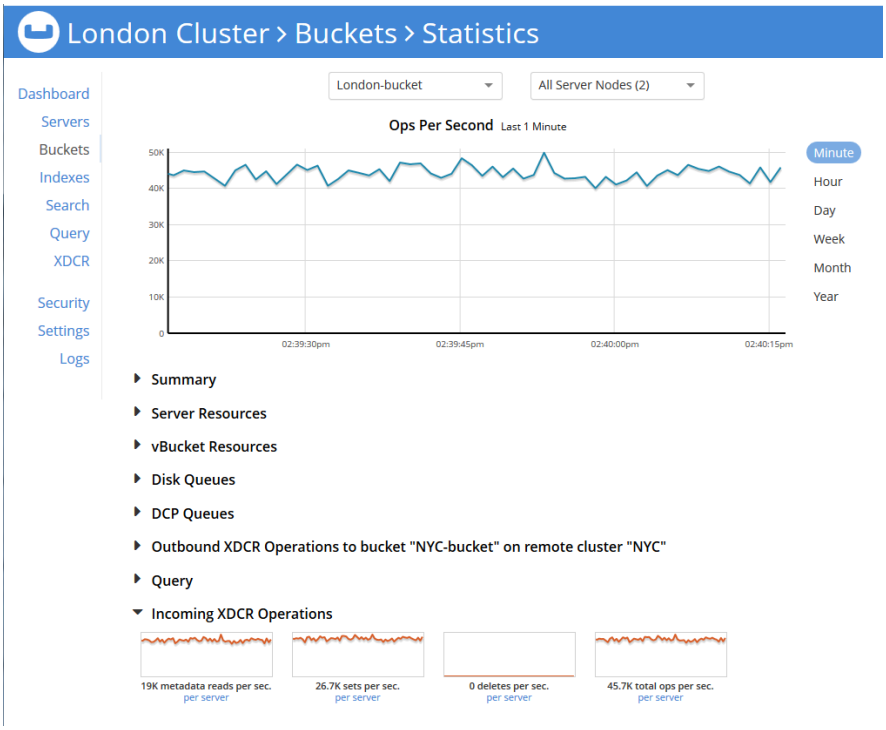
Note that in the above command if you increase the item size to just barely over the optimistic threshold, like to 300 bytes, then optimistic replication won't technically kick in as this is an approximate setting. So, in our case, we are increasing the item size well over the 256 byte setting to 10,000 bytes to make sure that optimistic replication does not kick in at all for these keys. To be more technically accurate, for checking the optimistic replication threshold, Couchbase checks the compressed size of an item on disk when deciding whether to send it optimistically or not.
Remember, we are writing 100,000 keys with the exact same key prefix, so we are actually doing updates to the existing 100,000 keys from the previous run of pillowfight.

Anyway, switch to the 6-node NYC cluster's Web UI and look at the "OUTBOUND XDCR OPERATIONS" to the London-bucket:



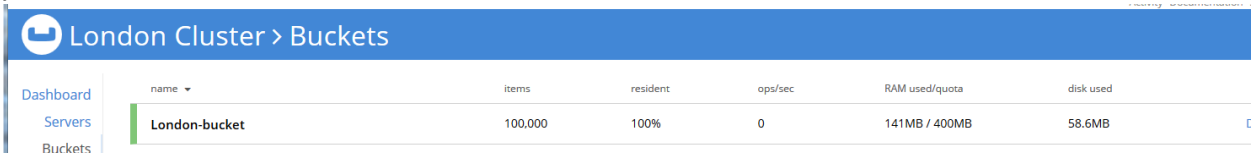
Notice in the screenshot above that the graph in the 2nd row, 4th column shows lower optimistic rate of replication. This is because there are now more metadata reads occurring on the NYC side!

Switch to the 2-node London cluster's Web UI and look at the "INBOUND XDCR OPERATIONS":



Observe here also that there are around 16-19K metadata reads per second happening. This is the 6-node NYC cluster doing metadata reads from London.

Stop the pillowfight command. (cntrl + C)
, remain in the 2-node London cluster's web UI and click on "Buckets link" at the side:



You will see a total of 100,000 items in the London-bucket. This is because we ran pillowfight in a loop with item count of 100,000 in this lab, the first 100,000 items written were actually updated.

Demonstrate that XDCR Replication keeps occurring to remaining nodes even after a node goes down and fails over:

Currently you should have bidirectional, encrypted replication set up between 6-node NYC and 2-node London. There are 100,000 keys with the prefix 'from_NYC' in buckets on both sides.

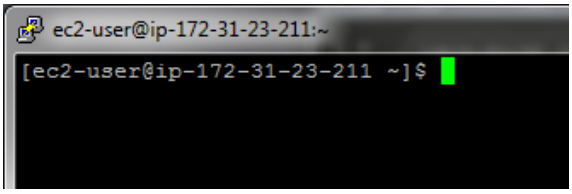
What do you think will happen if while you're inserting keys into the NYC cluster there is a node failure on one of the 2 nodes in London? If the 2nd node in London crashes, about half of the active vBuckets in London will be down and therefore half of the data from NYC can no longer be sent via XDCR to London.

But if we failover the node that crashed in London (2nd Node), then we would make the corresponding replicas on Node #1 in London active, thereby activating all 1024 vBuckets on the first London node.

In this section, you will then see that the items that could not be replicated temporarily while the 512 vBuckets in London were down, will suddenly start to be replicated to the newly promoted 512 vBuckets on the 1st node.
To demonstrate this concept we will use the vbucketkeygen tool to write 1 key to each of the 1024 vBuckets on the 6-node NYC cluster.

First, let's figure out how to use a combination of the Couchbase tools 'vbucketkeygen' and 'cbc create' along with the Linux tools curl, awk, echo and pipes to write exactly one key to each of the 1024 vBuckets in the NYC-bucket. This will essentially require us to write a simple script.

Switch to the App Server (black VM):



First pull the help menu for vbucketkeygen to learn how to use it:

```
[ec2-user@appserver ~]$ /opt/couchbase/bin/cbc-keygen --help

Usage:
  keygen [OPTIONS ...]

Output a list of keys that equally distribute amongst every vbucket
   -P  --password                  Bucket password [Default='']
   -u  --username                  Username [Default='']
```

```
   -Z  --config-cache                Path to cached configuration
[Default='']
   -U  --spec                        Connection string
[Default='couchbase://localhost/default']
       --truststorepath               [Default='']
       --certpath                    Path to server SSL certificate
[Default='']
       --keypath                     Path to client SSL private key
[Default='']
   -T  --timings                     Enable command timings
[Default=FALSE]
   -v  --verbose                     Set debugging output (specify
multiple times fo-
                                     r greater verbosity
[Default=FALSE]
       --dump                        Dump verbose internal state after
operations ar-
                                     e done [Default=FALSE]
   -y  --compress                    Turn on compression of outgoing
data (second ti-
                                     me to force compression)
[Default=FALSE]
   -D  --cparam  <OPTION=VALUE>      Additional options for
connection. Use -Dtimeou-
                                     t=SECONDS for KV operation
timeout [Default=]
       --keys-per-vbucket            number of keys to generate per
vbucket [Default=1]
   -?  --help                        this message
```

Notice in the yellow highlighted section above, that you have to give this tool a mapfile, then 2 parameters: the keys per vbucket and how many keys to generate. This essentially means that we can generate 10,000 keys and then if the 'keys per vbucket' parameter is set to 1, we will randomly pick 1,024 of the 10,000 keys and push 1 key into each of the 1,024 vBuckets. This would NOT MEAN that we are going to distribute 10,000 keys evenly into 1,024 vBuckets! Instead, it would mean that we are going to distribute 1,024 random keys (from a seed pool of 10,000 keys) evenly into 1,024 vBuckets.

Run the following command to provide the output from curl as the mapfile and then generate 10,000 keys and place 1 of those keys into each of the 1024 vBuckets. Provide this command with the public hostname of Node #1 (dark blue VM in the 6-node NYC cluster).

```
SKIP THIS STEP>>>>>>>>[ec2-user@appserver ~]$ curl -u
Administrator:couchbase -n
http://$NODE1:8091/pools/default/buckets/NYC-bucket |
/opt/couchbase/bin/cbc-keygen 1
```

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100 12275  100 12275    0     0   328k      0 --:--:-- --:--:-- --:--
:-- 1712k
key_0000008638 0
key_0000001149 1
key_0000000146 2
key_0000001320 3
key_0000000093 4
key_0000001365 5
key_0000000418 6
<output truncated>
```

Notice in the above output that key # ending 8638 would go into vBucket 0.

Now the above command does NOT actually push the keys into the vBucket. It is just a generator.

We have to pipe the output from the above command to series of other commands including 'cbc create' to actually push the keys into Couchbase.

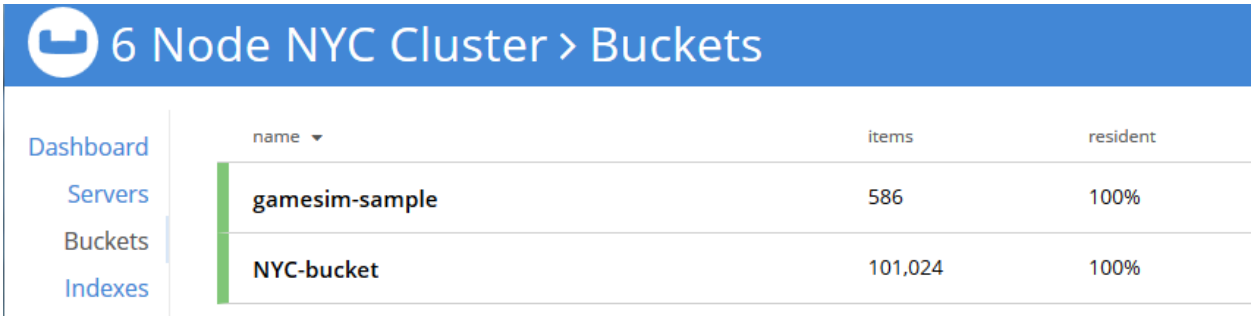
In the following command, we are inserting 1 key into each of the 1,024 vBuckets in the bucket NYC-bucket in the 6-node NYC cluster. We are going to prefix each key with AA.key#. Prefixing with "AA." will lexographically sort all of the keys in the Web UI's view, so it'll be easy to find all the AA keys from this generation. Provide this command with the public hostname of Node #1 (dark blue VM in the 6-node NYC cluster).

Note: the command in red will work if using libcouchbase 2.8.2 or better

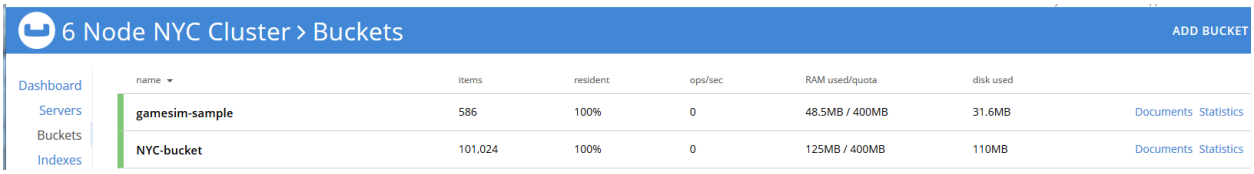
```
[ec2-user@appserver ~]$ for i in `curl -u Administrator:couchbase -n
http://$NODE1:8091/pools/default/buckets/NYC-bucket
|/opt/couchbase/bin/cbc-keygen -U couchbase://$NODE1/NYC-bucket --
keys-per-vbucket 1 |  awk {'print $1'}`;do echo -n "value" |
/opt/couchbase/bin/cbc-create -U couchbase://$NODE1/NYC-bucket
AA.$i;done

AA.key_0000000877   Stored. CAS=0x158ead9bacd70000
AA.key_0000001878   Stored. CAS=0x158ead9bad980000
AA.key_0000001468   Stored. CAS=0x158ead9bae5b0000
AA.key_0000000139   Stored. CAS=0x158ead9baf220000
AA.key_0000009648   Stored. CAS=0x158ead9bafe20000
AA.key_0000000350   Stored. CAS=0x158ead9bb0a30000
AA.key_0000000832   Stored. CAS=0x158ead9bb1610000
<output truncated>
```

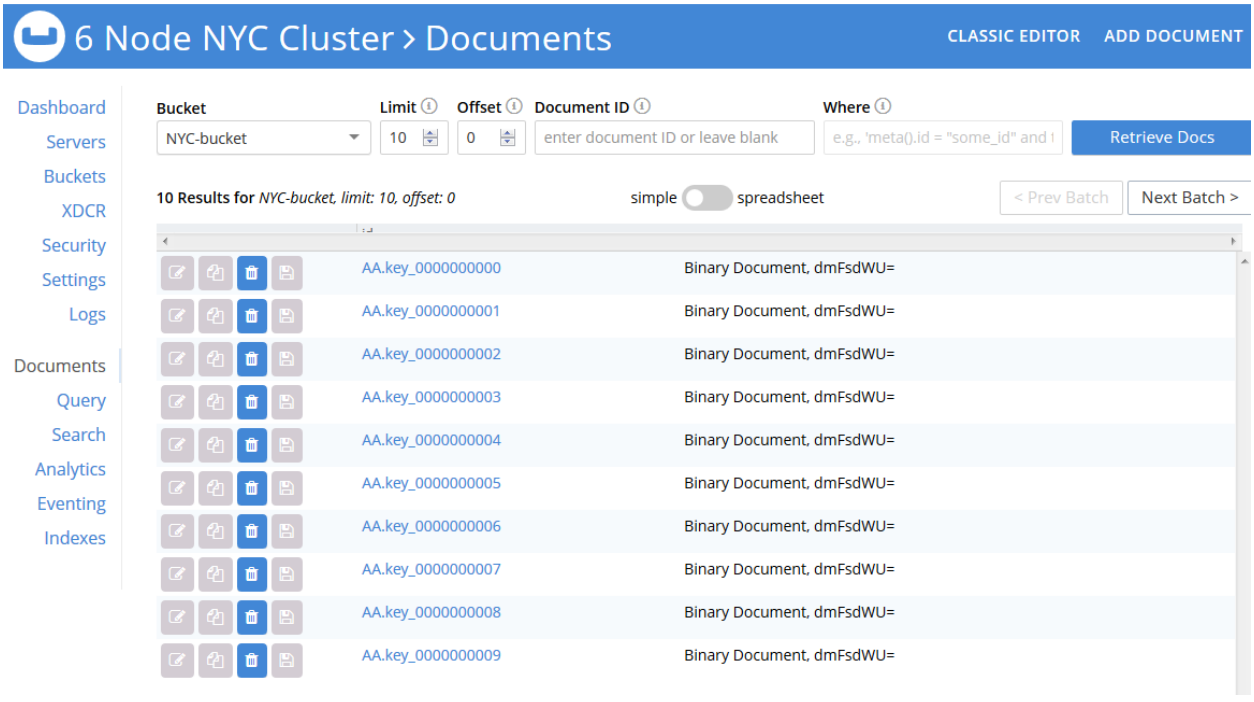
If you now switch to the browser tab for the 6-node NYC cluster and refresh the page, you will see 101,024 keys in the NYC-bucket. We just added the last 1,024 keys.



Try clicking on Documents for the "NYC-bucket" to actually display the items:



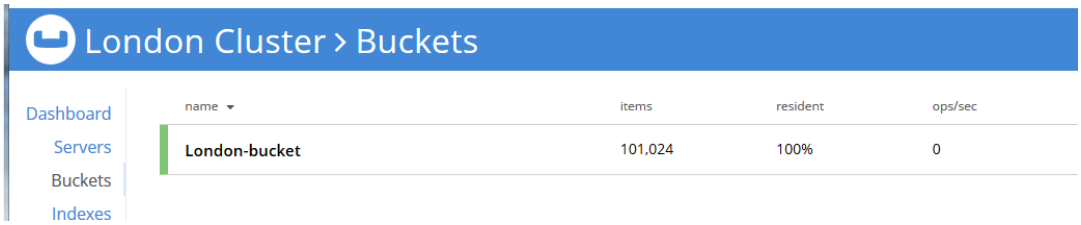
Here you will see the keys starting with "AA." that we just added. Change the dropdown for item count on the page in the bottom left corner from the default of 10 to 100:



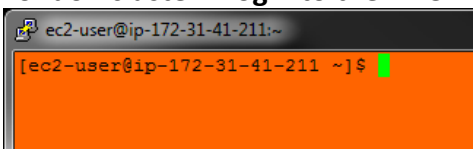
You may be surprised that all of the keys you see are in sequential order. For example the screenshot above shows keys 00 – 05 all in order. This does not mean that the script we ran inserts key 0 into vBucket 0 and key 24 into vBucket 24!

Try clicking the page to page 10 and you will start to see that the AA. keys can get higher than 1024:

Did the new 1,024 keys also get replicated to London? Switch to the browser tab for the 2-node London cluster, refresh the page and look at the Data Buckets page to see if the item count has increased:



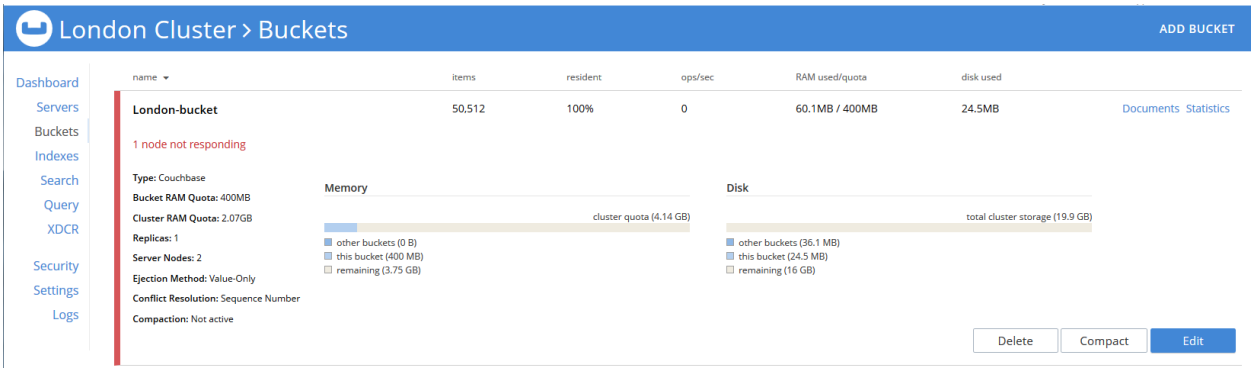
It has increased. So replication is also working as expected.
Now we will do something interesting. Let's cause a simulated failure on the 2nd node in the London cluster. Log into the XDCR Node #2 in London (Orange VM/Couchbase08):



We're going to pretend that the Couchbase service has crashed here… or pretend that the entire machine has crashed. We'll simulate this by simply stopping the Couchbase service here:

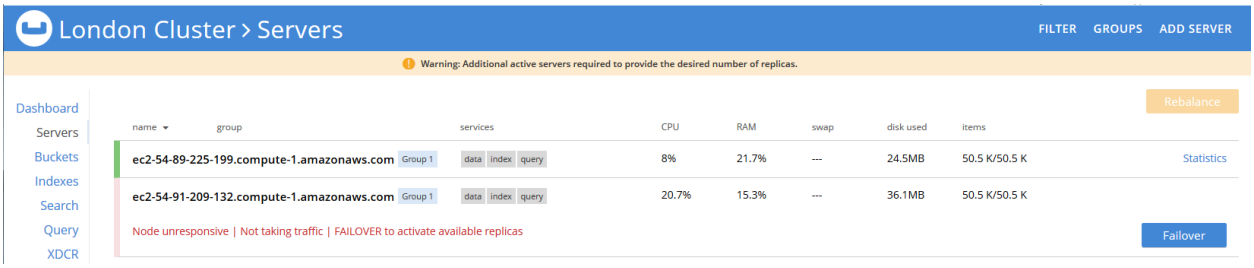
```
[ec2-user@couchbase08 ~]$ sudo systemctl stop couchbase-server
Stopping couchbase-server
[ec2-user@couchbase08 ~]$ sudo systemctl status couchbase-server
couchbase-server is not running
```

Switch back to the Web UI for the 2-node London cluster and under the Data Buckets page, notice that the bucket is now red, signaling that half of the data is now unavailable b/c half of our London cluster is down.



Notice in the above screenshot also that the current Item Count is only 50,512 since half the active vBuckets are now missing! Keep this # in mind.

Click on Server link at the side and notice that the 2nd machine is in a down state:



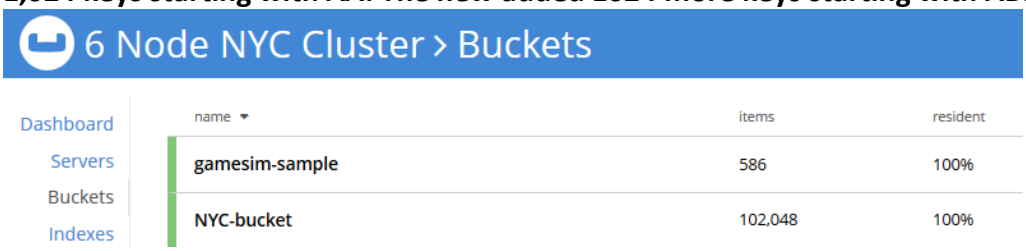
Switch back to the App Server (black VM). Now, let's try to write 1,024 keys into the 6-node NYC cluster and see how many of them successfully replicate to the 1-node London cluster. This time prefix the keys with "AB." b/c this is the 2nd time we're writing 1,024 keys. Still provide this command the public hostname of Node #1 in the 6-node NYC cluster.

```
[ec2-user@appserver ~]$ for i in `curl -u Administrator:couchbase -n
http://$NODE1:8091/pools/default/buckets/NYC-bucket
|/opt/couchbase/bin/cbc-keygen -U couchbase://$NODE1/NYC-bucket --
keys-per-vbucket 1 |  awk {'print $1'}`;do echo -n "value" |
/opt/Couchbase/bin/cbc-create -U couchbase://$NODE1/NYC-bucket
AB.$i;done
```

```
AB.key_0000000285   Stored. CAS=0x158eaef057590000
AB.key_0000000877   Stored. CAS=0x158eaef057fc0000
AB.key_0000001878   Stored. CAS=0x158eaef058980000
AB.key_0000001468   Stored. CAS=0x158eaef059310000
AB.key_0000000139   Stored. CAS=0x158eaef059cd0000
AB.key_0000009648   Stored. CAS=0x158eaef05a670000
```

```
AB.key_0000000350    Stored. CAS=0x158eaef05b020000
AB.key_0000000832    Stored. CAS=0x158eaef05b9d0000
<output truncated>
```

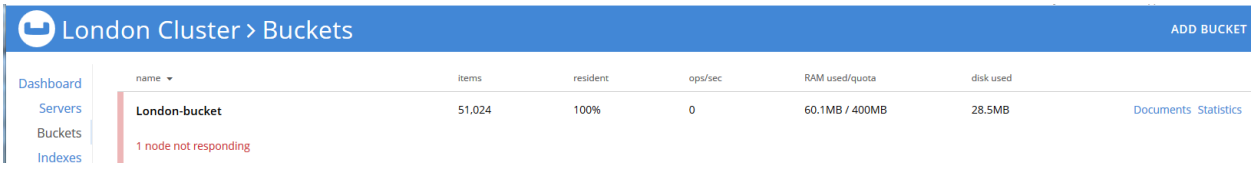
Switch to the browser tab for the 6-node NYC cluster, go to the Data Bucket page, refresh it and notice that the item count is 102,048. This makes sense. Originally the bucket had 100,000 keys (from the pillowfight command we ran in the prior section). Then we added 1,024 keys starting with AA. The new added 1024 more keys starting with AB.



Switch to the browser tab for the 2-node London cluster (which is down to 1 healthy node), go to the Bucket page, refresh it and notice that the item count is 51,024.

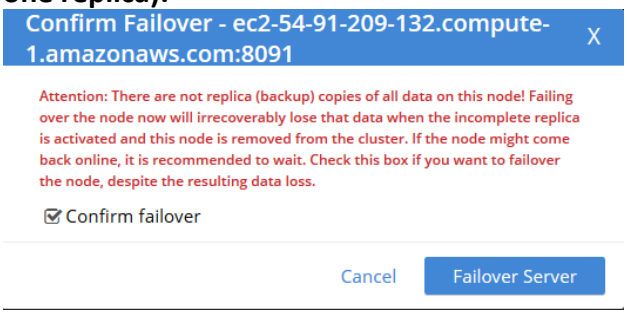
This makes sense also. Remember that after we lost 1 node in London, the item count went down from 101,024 to 50,512 (it became half). Then we replicated 512 or so more keys with the prefix AB from NYC -> London. So 50,512 + 512 = 51,024. The issue here is that there are 512 keys in a sort of "pending replication" state in the NYC cluster. The NYC cluster cannot send them to London b/c half the active vBuckets in London are missing. We have to Fail Over the down node in London to promote the replicas on the remaining node. Then the updated cluster map from London (with one node owning all 1,024 vBuckets) will be sent to all nodes in NYC and the 4 NYC nodes will push out the last 512 keys with prefix AB over to the London 1-node cluster.

Note that a Rebalance is not required in London to get the cluster map updated on the NYC side.



Switch back to the London cluster's Web UI, click on "Servers" at the side, then click the "Fail Over" button next to the down node to promote its replicas on the remaining node.

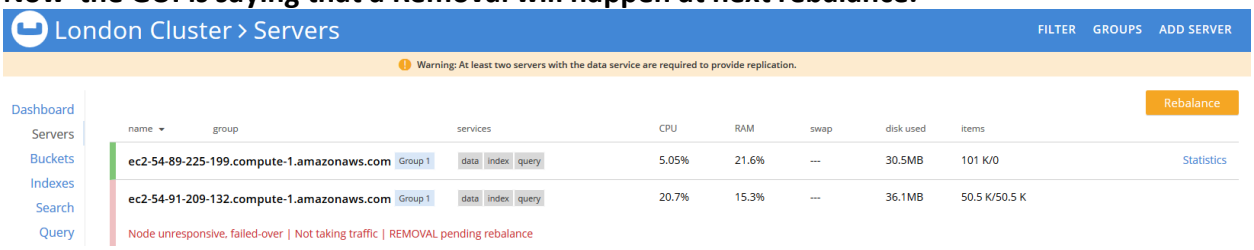
Remember that we configured the London-bucket with 1 replica (so 2 copies, one active and one replica).



Side note: The Fail Over cannot be configured to occur automatically in London, since there are only 2 nodes in the cluster. There have to be at least 3 nodes in a Couchbase cluster before auto-failover can be enabled. This is because with a 2-node cluster, if failover was automatic, then a split-brain condition could have occurred. (Well, technically it is possible to configure auto-failover in a 2-node cluster, but it won't get triggered if a node goes down. If you grow the same cluster to 3+ nodes, then auto-failover will start to get triggered when a node goes down.)

On the pop-up, place a check next to "Please confirm Failover" and click "Fail Over":

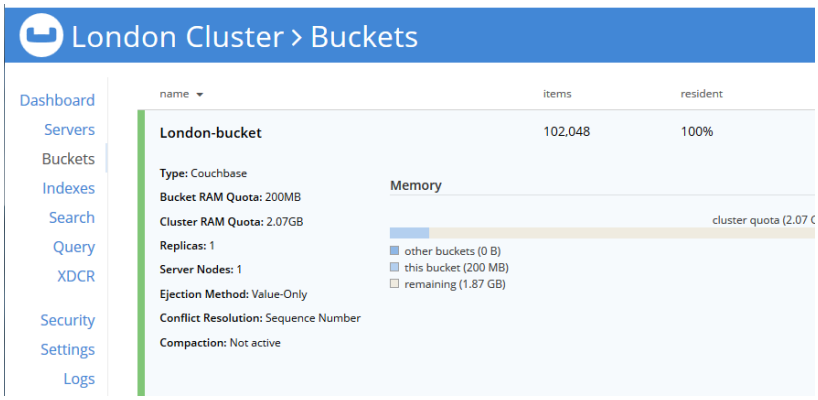
Now the GUI is saying that a Removal will happen at next rebalance.



At this point, the updated cluster map from London (that one node owns all 1,024 vBuckets) has been sent to all 4 data nodes in NYC and the 4 data NYC nodes have started pushing out the last 512 keys with prefix AB over to the London 1-node cluster.

In this case, the Web UI is simply asking you to run a Rebalance to remove down node from the cluster. If you had clicked on Rebalance, no rebalancing would actually occur. The rebalance is being requested to permanently remove the down node from the vBucket cluster map.

Let's verify this.
In the London (now 1-node) cluster's Web UI, click on Buckets link at the side and look at the item count:



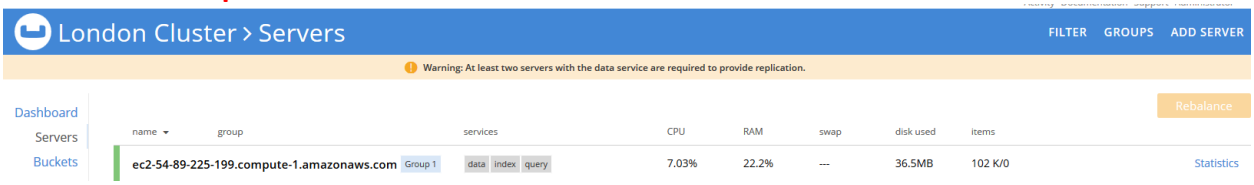
The item count now matches what we see in the 6-node NYC cluster. Perfect! (*If you don't see the same item count yet, you may need to wait 1 – 2 mins for XDCR to finish replicating and then refresh the page*)

This proves that as soon as you Failed Over the down node in London, the 6-node NYC cluster has the 512 vBuckets become active again on the remaining London node and then the 4 nodes in NYC started to replicate out those last 512 keys to London.

The steps we completed above demonstrate that if we had auto-failover configured in London with 3+ nodes in the London cluster, then even if we lose a node, replication would eventually resume automatically after the auto-failover is triggered.

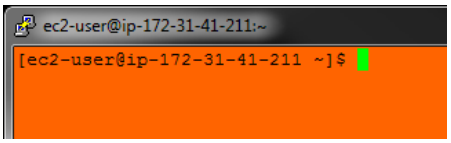
Before concluding this section, let's Rebalance the London cluster so that the Web UI shows only 1 node, then we'll re-add the down node back into the cluster and do another rebalance so that the London cluster becomes a healthy 2 node cluster again.

Switch to the browser tab for the London cluster (currently 1 node), and click on "Server Nodes" at the top and then hit Rebalance:



The rebalance will finish in just a few seconds as there is really nothing to rebalance:

Log into the XDCR Node #2 in London (Orange VM):

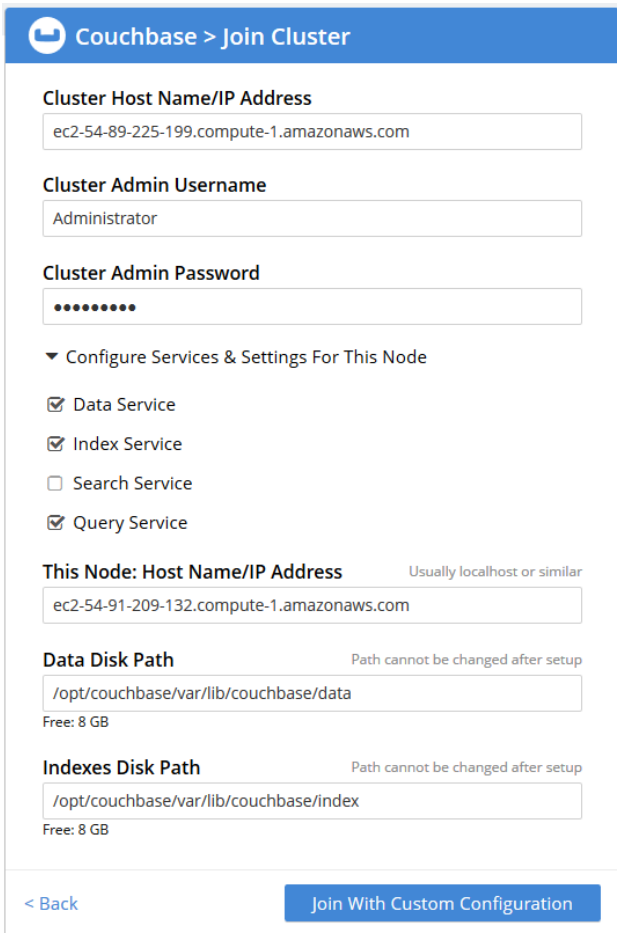


This is the node in London that we had previously stopped the Couchbase service on. Verify that the Couchbase service is still stopped and then start it again:

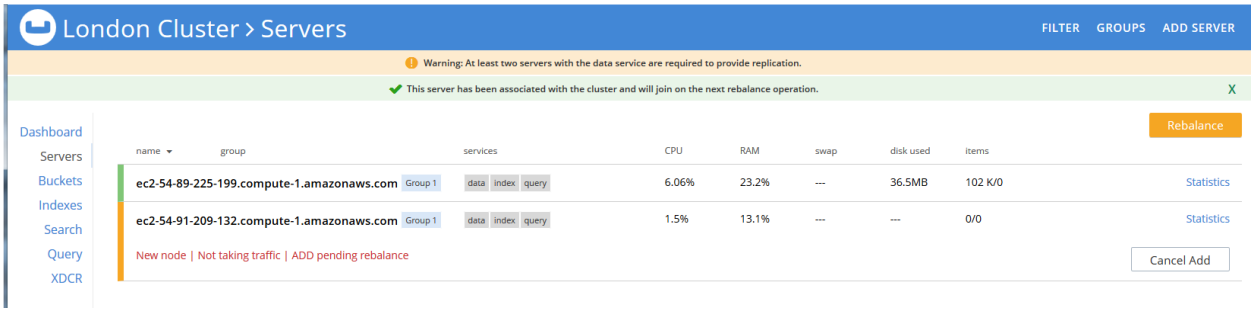
```
[ec2-user@ip-172-31-41-211 ~]$ sudo systemctl status couchbase-server
couchbase-server is not running


[ec2-user@ip-172-31-41-211 ~]$ sudo systemctl start couchbase-server
Starting couchbase-server                                  [  OK  ]
```

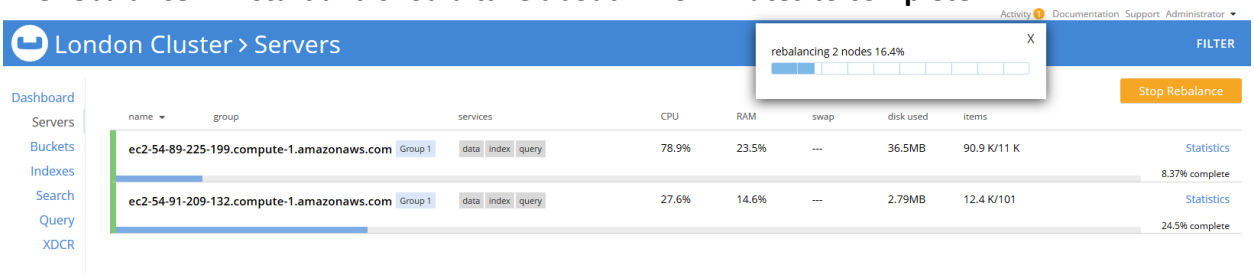
Add the server back to the two node London cluster using the Join method covered in Lab #3



You will see a #1 under Pending Rebalance now as the new node has been added to the cluster, but the active and replica vBuckets still have to be redistributed across the two nodes. Click on Rebalance:



The rebalance will start and should take about 2 – 3 minutes to complete:



This concludes Lab #7.