CS300 Couchbase NoSQL Server Administration

Lab 8 Exercise Manual

Release: 4.5

Revised: July 26th, 2016

Lab #8: Performance

Objective: Understanding performance metrics and benchmarking distributed clusters can be a very complex topic as it could involve the linux kernel, disk subsystem, network lines and user land software algorithms. However, in this 90 minute lab, we will focus on studying performance purely from Couchbase's perspective. In a production environment, many other factors will come into play when diagnosing low performing clusters.

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

- Performance Demo #1: Understanding high water mark and low water mark
- Understand how ejection works
- Learn about the Not-Recently-Used metadata setting
- Learn about the item pager
- Become familiar with how Couchbase manages memory
- Use cbstats and the Web UI to read the high and low water mark settings
- Performance Demo #2: Disk reads vs RAM reads
- Understand how different traffic patterns require different Couchbase settings
- Learn what happens when most of the reads are served from disk vs. RAM
- Understand the 'resident %' in memory metric for active and replica items
- Learn how to detect out of memory (OOM) errors from the performance charts
- Performance Demo #3: Displaying metrics via cbstats and studying item expiration
- Display timing metrics with "cbstat timings"
- Learn about item expiration in Couchbase
- Learn about compaction

Performance Demo #1: Understanding high water mark and low water mark:

Couchbase Server actively manages the data stored in a caching layer; this includes the information which is frequently accessed by clients and which needs to be available for rapid reads and writes. When there are too many items in RAM, Couchbase Server removes certain data to create free space and to maintain system performance. This process is called "working set management" and the set of data in RAM is referred to as the "working set".

Ejection & Keeping keys + metadata in RAM

In general the working set consists of all the keys, metadata, and associated documents which are frequently used require fast access. The process the server performs to remove data from RAM is known as “Value ejection”. When the server performs this process, it removes the document, but not the keys or metadata for the item. Keeping keys and metadata in RAM serves three important purposes in a system:

- Couchbase Server uses the remaining key and metadata in RAM if a request for that key comes from a client. If a request occurs, the server then tries to fetch the item from disk and return it into RAM.
- The server can also use the keys and metadata in RAM for “miss access”. This means that it is quickly determine whether an item is missing and if so, perform some action, such as add it.
- Finally, the expiration process in Couchbase Server uses the metadata in RAM to quickly scan for items that are expired and later remove them from disk. This process is known as the “expiry pager” and runs every 60 minutes by default.

Not-Frequently-Used Items

All items in the server contain metadata indicating whether the item has been recently accessed or not. This metadata is known as not-recently-used (NRU). If an item has not been recently used, then the item is a candidate for ejection if the high water mark has been exceeded. When the high water mark has been exceeded, the server evicts items from RAM.

Couchbase Server provides two NRU bits per item and also provides a replication protocol that can propagate items that are frequently read, but not mutated often.

NRUs are decremented or incremented by server processes to indicate an item is more frequently used, or less frequently used. Items with lower bit values have lower scores and are considered more frequently used. The bit values, corresponding scores and status are as follows:

Binary NRU	Score	Working Set Replication Status (WSR)	Access Pattern	Description
00	0	TRUE	Set by write access to 00. Decremented by read access or no access.	Most heavily used item.
01	1	Set to TRUE	Decremented by read access.	Frequently access item.
10	2	Set to FALSE	Initial value or decremented by read access.	Default for new items.
11	3	Set to FALSE	Incremented by item pager for eviction.	Less frequently used item.

There are two processes which change the NRU for an item:

- A client reads or writes an item, the server decrements NRU and lowers the item's score
- A daily process which creates a list of frequently-used items in RAM. After this process runs, the server increments one of the NRU bits.

Because the two processes changes NRUs, they also affect which items are candidates for ejection.

Couchbase Server settings can be adjusted to change behavior during ejection. For example, specify the percentage of RAM to be consumed before items are ejected or specify whether ejection should occur more frequently on replicated data than on original data. Couchbase recommends that the default settings be used.

Item Pager

The item pager process, which runs periodically, removes documents from RAM and retains the item's key and metadata. If the amount of RAM used by items reaches the high water mark (upper threshold), both active and replica data are ejected until the memory usage (amount of RAM consumed) reaches the low water mark (lower threshold). Ejections of active and replica data occur with the ratio probability of 40% (active data) to 60% (replica data) until the memory usage reaches the low watermark. Both the high water mark and low water mark are expressed as a percentage amount of RAM, such as 80%.

Both the high water mark and low water mark can be changed by providing a percentage amount of RAM for a node, for example, 80%. Couchbase recommends that the following default settings be used:

High water mark: 85%
Low water mark: 75%

The item pager ejects items from RAM in two phases:

Phase 1: Eject based on NRU. Scan NRU for items and create list of all items with score of 3. Eject all items with a NRU score of 3. Check RAM usage and repeat this process if usage is still above the low water mark.

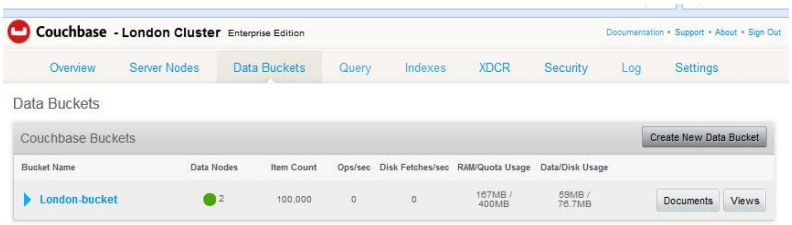
Phase 2: Eject based on Algorithm. Increment all item NRUs by 1. If an NRU is equal to 3, generate a random number and eject that item if the random number is greater than a specified probability. The probability is based on current memory usage, low water mark, and whether a vBucket is in an active or replica state. If a vBucket is in active state the probability of

ejection is lower than if the vBucket is in a replica state. The default probabilities for ejection from active of replica vBuckets is as follows:

Now that we've covered some important Couchbase memory management concepts, let's see them hands on in a demonstration.

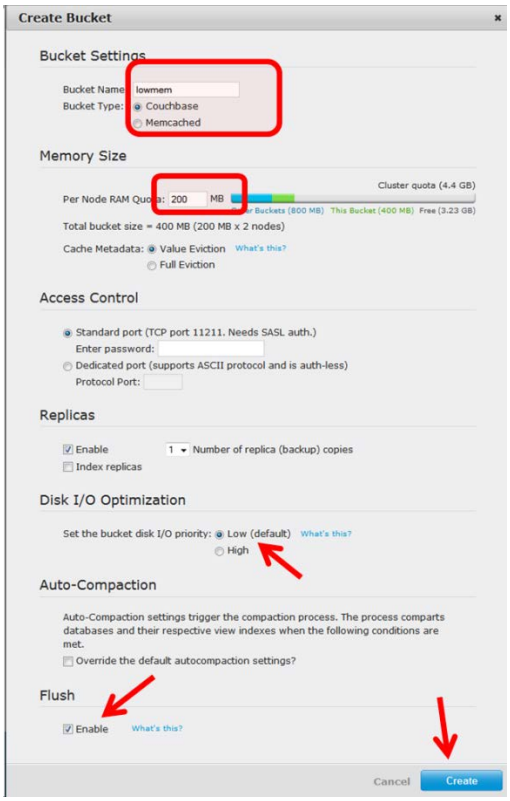
For the first demo, let's create a new bucket on the 2-node London cluster named 'lowmem' with a 200 MB per node RAM quota. Over 2 nodes, this bucket will have a RAM quota of 400 MB. Then we will use pillowfight to insert items in an infinite loop of size 2000 bytes each to exhaust the 400 MB of memory. Once the RAM usage hits the high water mark, we will see that the ejection process gets triggered and some of the working set in RAM gets moved to disk.

Switch to the browser tab for the 2-node London cluster, click on Data Buckets at the top and then click "Create New Data Bucket":



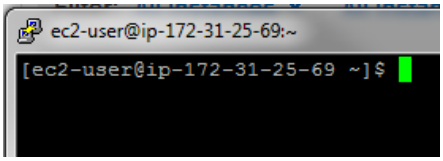
Use the following settings on the Create Bucket screen (only parameters in red should need to be altered):

Bucket Name: lowmem
Bucket Type: Couchbase
Per Node RAM Quota: 200 MB
Cache Metadata: Value Ejection
Access Control: Standard port (no password)
Replicas: Place a check next to Enable and set the # to 1
Disk I/O optimization: Low
Auto-Compaction: DO NOT place a check to override the default autocompaction settings
Flush: Enabled
Click Create



You should now see the 'lowmem' data bucket:

Switch over to the App Server (black VM) and let's query the disk readers/writers setting for a bucket:



Run the following cbstats command against the public hostname for Node #1 (red VM) in the 2-node London cluster:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-
1.amazonaws.com:11210 -b lowmem all | egrep 'high_wat|low_wat'
 ep_mem_high_wat:                178257920
 ep_mem_low_wat:                 157286400
```

Note that water marks are given in bytes.

The high_wat mark translates into 170 MB.

The low-wat mark translates into 150 MB.

The above settings are per node. So, for the entire lowmem bucket (on 2 London nodes), the high water mark is 170 MB * 2 = 340 MB. Therefore the low water mark across the bucket is 300 MB.

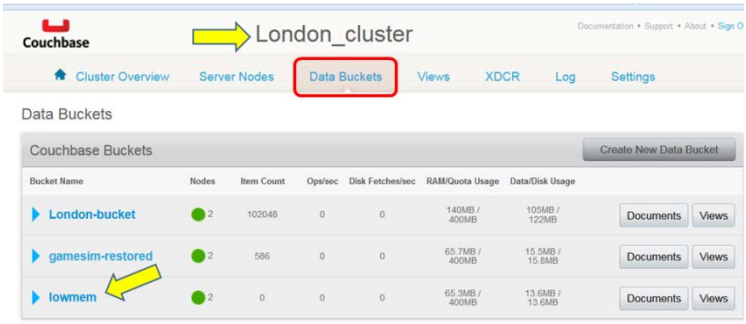
Remember that the item pager process ejects items from RAM when too much space is being taken up in RAM. Ejection means that documents are removed from RAM but the key and metadata remain.

If the amount of RAM used by items reaches the high water mark (upper threshold), both active and replica data are ejected until the memory usage (amount of RAM consumed) reaches the low water mark (lower threshold).

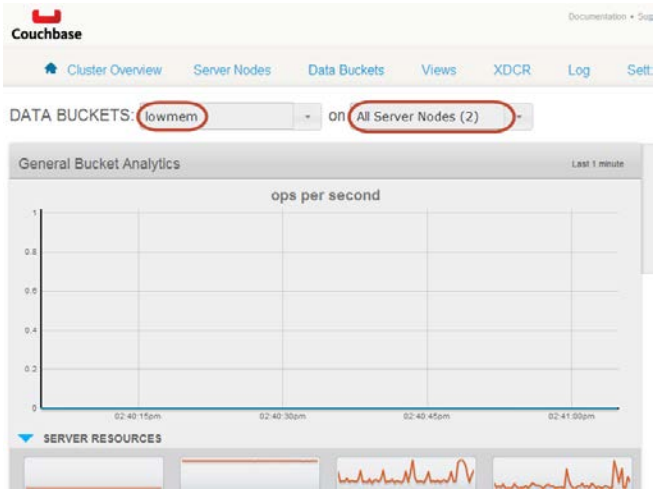
The server determines that items are not frequently used based on a not-recently-used (NRU) value. There a few settings you can adjust to change server behavior during the ejection process. In general, we do not recommend you change ejection defaults for Couchbase Server unless you are required to do so.

Anyway, let's verify the high and low water mark settings in the Web UI as well.

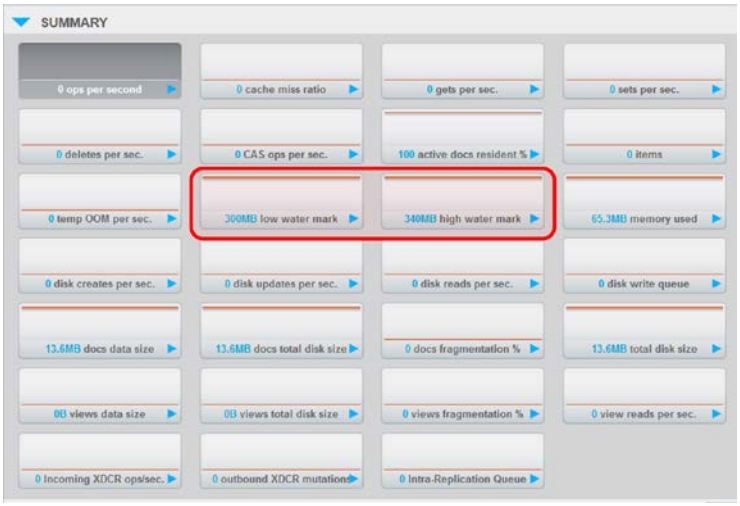
Switch to the browser tab for the 2-node London cluster, click on Data Buckets at the top and then click on the bucket named "lowmem":



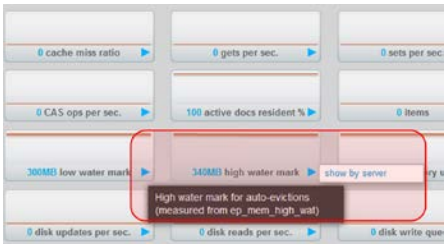
The performance and metrics UI for the lowmem bucket across 2 nodes will appear:



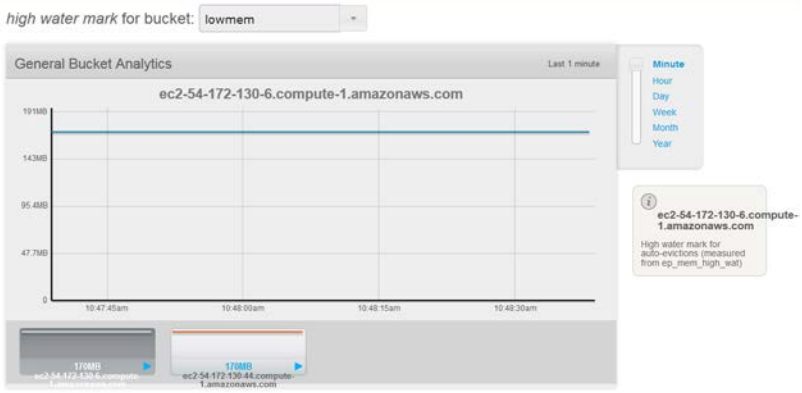
Scroll down on the performance page until you see the 'Summary' metrics, then locate the high water mark chart. Notice that the high water mark across 2 nodes is set to 340 MB and the low water mark is set to 300 MB across the nodes:



Click the blue arrow next to high water mark and choose 'show by server':



A new chart will appear show the high water per node.



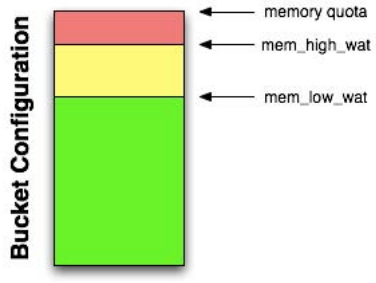
Notice that this metric is queried via the ep_mem_high_wat setting. This is the same setting that we also queried earlier using the cbstats command. The cbstats command had said that the high water mark was 170 MB.

Let's change the high water mark settings for the 'lowmem' bucket. Note that we are only calling this bucket 'lowmem' because we allocated it only 400 MB of RAM total (200 MB on each node). The 'lowmem' name has nothing to do with the low water mark!

Warning: Be aware that this tool (cbstats) is a per-node, per-bucket operation. To perform this operation for an entire cluster, perform the command for every node/bucket combination that exists for that cluster.

Per the output of the cbstats command, the high water mark is 170 MB. Let's change this to 172 MB for the lowmem bucket for both nodes just to see how it is done.

This screenshot shows the basic memory configuration for a bucket:



Switch to the App Server (black VM). Run the following cbepctl command against Node #1's public hostname in the 2-node London cluster (Note that 180355072 bytes translates to 172 MiB):

```
[ec2-user@ip-172-31-23-211 ~]$ cbepctl ec2-54-86-94-x.compute-
1.amazonaws.com:11210 -b lowmem set flush_param mem_high_wat 180355072
setting param: mem_high_wat 180355072
set mem_high_wat to 180355072
```

Then verify the setting:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-
1.amazonaws.com:11210 -b lowmem all | egrep 'high_wat|low_wat'
 ep_mem_high_wat:                    180355072
 ep_mem_low_wat:                     157286400
```

Okay, great, now let's make the same change on Node #2 in the 2-node London cluster. Provide the hostname for Node #2 in the London cluster here:

```
[ec2-user@ip-172-31-23-211 ~]$ cbepctl ec2-54-86-96-x.compute-
1.amazonaws.com:11210 -b lowmem set flush_param mem_high_wat 180355072
setting param: mem_high_wat 180355072
set mem_high_wat to 180355072
```

Then verify the setting:

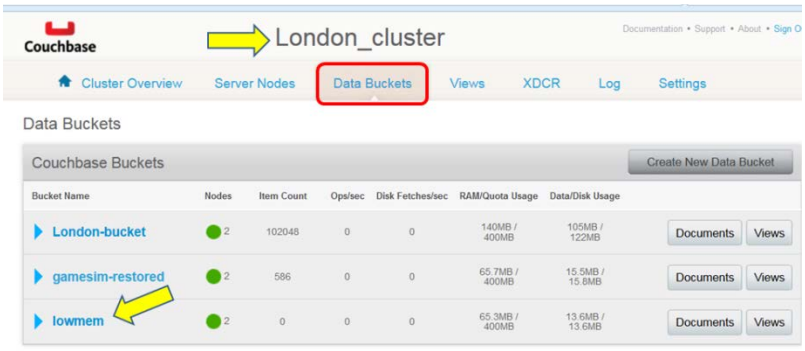
```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-96-x.compute-
1.amazonaws.com:11210 -b lowmem all | egrep 'high_wat|low_wat'
 ep_mem_high_wat:                    180355072
 ep_mem_low_wat:                     157286400
```

Next, switch to the Web UI for the 2-node London cluster and you should see the change take effect on the same page that you had loaded previously. The high water mark for both nodes in the GUI should now read 172 MiB:

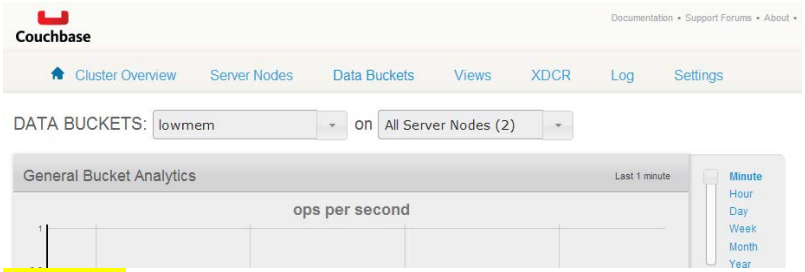
Finally, let's demonstrate how these two settings actually work in real life!

We will now use pillowfight to push writes the size of 2000 bytes in a loop into the lowmem bucket in the 2-node London cluster.

But first, load of the correct graphs page so that we can come back to this page quickly once the pillowfight writes start. Switch to the browser tab for the 2-node London cluster, click Data Buckets at the top menu and then click on the bucket "low mem":



The performance metrics page for the lowmem bucket across 2 server nodes will now load:

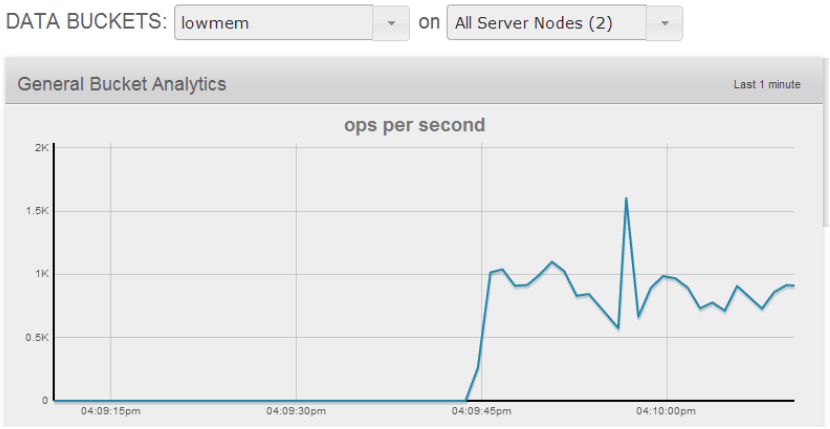


Warning: There are a lot of steps to complete quickly over the next 5 or 6 pages to see the expected behavior from this demonstration. It may be best to first read through the next 6 pages and then come back here to start running pillow fight. That way you know what to quickly do next after you pillow fight begins!

Switch to the App Server (black VM). The following command will write 100,000 items of size 2000 bytes into the 2-node cluster using 2 threads. Run the following pillowfight command against Node #1's public hostname in the 2-node London cluster (*Remember to not copy + paste commands that span multiple lines like this one!*):

```
[ec2-user@ip-172-31-23-211 ~]$ cbc-pillowfight -U couchbase://ec2-54-
172-130-44.compute-1.amazonaws.com/lowmem  --num-items=100000  --set-
pct=100 --min-size=2000 --max-size=2000 --timings --num-cycles=10 -l
Creating instance 0
[1416338162.232820] Running. Press Ctrl-C to terminate.....
```

Quickly switch to the web UI and you should see write operations coming into the cluster:



Scroll down to the Summary section and notice the following 4 graphs:

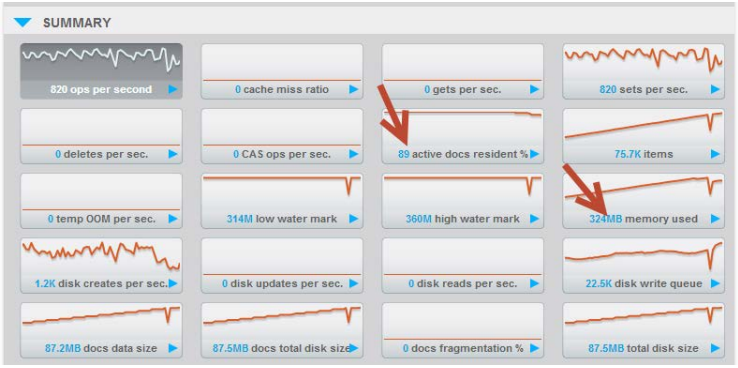
- The cluster is getting about .6-1.5 K ops per second
- 100% of the active docs are resident in memory
- The high water mark is set to 344 MiB in the web UI
- In the screenshot below, the amount of memory used across both nodes is 238 MB.

What you should specifically be watching for now is when the 'memory used' hits the same # as the 'high water mark' you should see the 'active docs resident %' go down.

So, when the 'mem used' hits 344 MB, you will see the 'active docs resident %' start to get lower than 100%. This is because at that point the pillow fight app is starting to fill up the 400 MB we have allocated for this bucket across 2 nodes. When the RAM fills up to the high water mark (344 MB), Couchbase will start to eject items from memory, so the 'active docs resident %' will start to go down.

Also, scroll down and expand the VBUCKET RESOURCES section of the metrics and notice that so far there are 0 ejections per second. Once the high water mark hits, you will start to see ejections occurring as items are removed from the RAM cache:

In this screenshot, the memory used has just hit 344 MB, so the active docs resident is now lower than 100%. The items will continue to be ejected from memory until the memory used hits the 'low water mark':

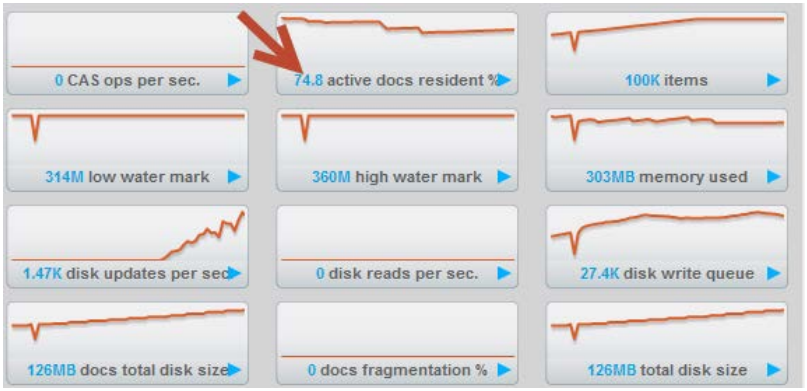


As the data is being ejected, quick scroll down to the VBUCKET RESOURCES section and notice that now the counter for 'ejections per second is increasing':

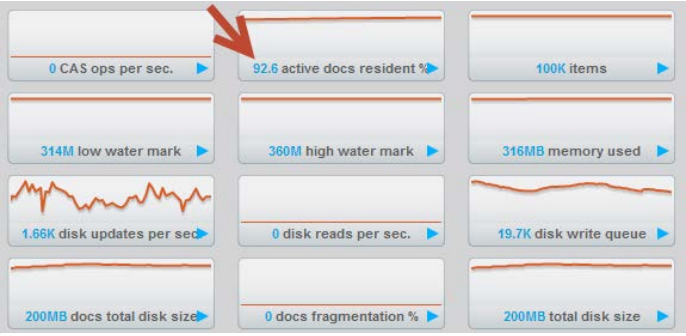
Notice that the above metrics also show you the amount of 'user data in RAM' and the amount of 'metadata in RAM'.

The # of ejections per second will not be a sustained linear line, but rather more sporatic. As the high water mark is hit, the ejections kick off until the low water mark is reached. Then the ejections stop (until the next time that the high water mark is hit).

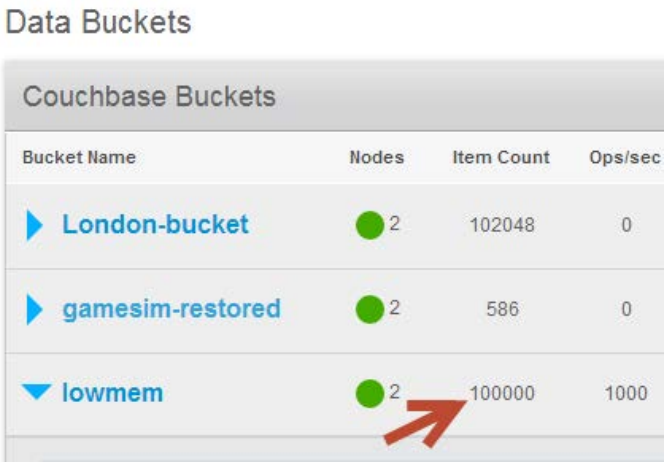
As time continues, you will see the 'active docs resident %' keep lowering as items are ejected from the RAM cache:



Interestingly, after another minute or so, you'll see the 'active docs resident %' actually increase again:



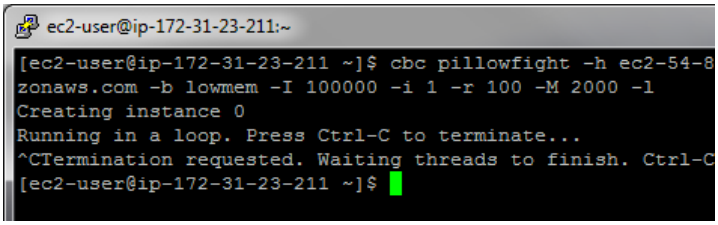
The 'active docs resident %' is increasing b/c pillow fight has already written the original 100,000 items to the lowmem bucket:



Now pillow fight is simply updating the data already in RAM with updated items. When this happens eventually the high water mark will be hit again. Then there's a 60% chance that a given replica will be ejected from RAM, but only a 40% chance that an active item will be ejected. This will continue until the low water mark is reached. Then as more updates keep occurring, the high water mark is reached again and more items are ejected. In each round more replicas will be ejected than active items… so eventually the RAM will get filled with just active items.

Eventually, almost 99% of the active items will be stored in the RAM cache again:

Switch over to the App Server (black VM) and hit <CTRL> + C to stop the pillowfight command:



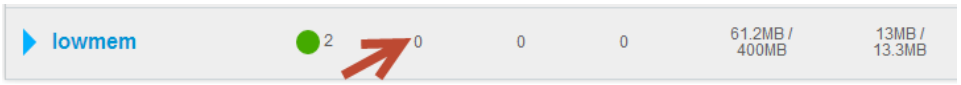

```
[ec2-user@ip-172-31-23-211 ~]$ cbc pillowfight -h ec2-54-86-94-
157.compute-1.amazonaws.com -b lowmem -I 100000 -i 1 -r 100 -M 2000 -l
Creating instance 0
Running in a loop. Press Ctrl-C to terminate...
^CTermination requested. Waiting threads to finish. Ctrl-C to force
termination.
[ec2-user@ip-172-31-23-211 ~]$
```

Next, use the Couchbase CLI tool to flush the 'lowmem' bucket and delete all of the 100,000 items that we wrote in there. Provide this command the public hostname of Node #1 in the 2-node London cluster. Also type Yes to accept the warning:

```
[ec2-user@ip-172-31-23-211 ~]$ couchbase-cli bucket-flush --
cluster=ec2-54-86-94-157.compute-1.amazonaws.com:8091 --
user=Administrator --password=couchbase2 -b lowmem
Running this command will totally PURGE database data from disk.Do you
really want to do it? (Yes/No)Yes <hit enter>
Database data will be purged from disk ...
 SUCCESS: bucket-flush
[ec2-user@ip-172-31-23-211 ~]$
```

After you enter Yes, it will take about 15 – 30 seconds before you'll see the SUCCESS message.

The UI should also now show the lowmem bucket as empty:



Performance Demo #2: Disk Reads vs RAM Reads:

As a Couchbase Server administrator, it is critical to understand the profile of your application traffic so that you can configure settings in Couchbase appropriately. Some of the most important traffic patterns to understand are:

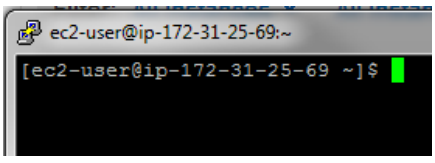
- the % of reads vs. writes vs. deletes vs. updates from the application
- does the application traffic spike during the day and die down during the night? (if so it may be best to run compaction in the middle of the night
- does the application read the same hot items repeatedly? If so, you will want to make sure these are served directly out of RAM (working set)
- When reads are occurring, what % of reads are going to disk vs. being read from RAM?

In this section, we will try to answer the last question of reads from disk vs. reads from RAM.

First, let's use pillowfight to generate a specific type of data to fill the 'lowmem' bucket. What we want to do specifically is have only 10% of our items in the bucket in RAM and the other 90% on disk. This way almost all of the reads will be forced to go to disk for the data.

The way we'll achieve this is by pushing 800 items of size 1 MB into the 'lowmem' bucket. Remember that this bucket's dynamic RAM Quota for cache is only 400 MB. Also, we have 1 replica configured for this bucket, so there will be contention to also store some of those replicas in the 400 MB RAM space.
Let's get started!

Switch over to the App Server (black VM):



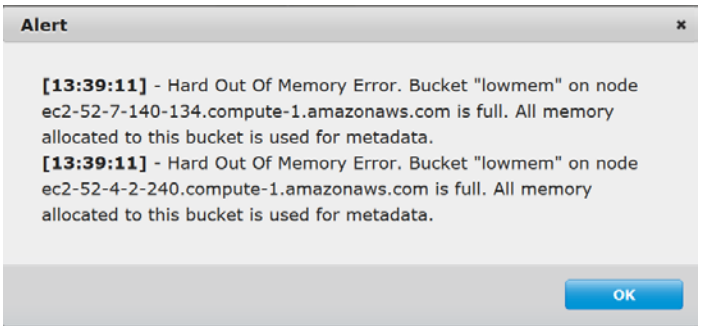
Run the following pillowfight command against the public hostname of Node #1 in the 2-node London cluster:

```
[ec2-user@ip-172-31-23-211 ~]$ cbc-pillowfight -U couchbase://ec2-54-172-130-44.compute-1.amazonaws.com/lowmem  --num-items=800 --set-pct=100 --min-size=1048575 --max-size=1048575 --timings --num-cycles=10
```

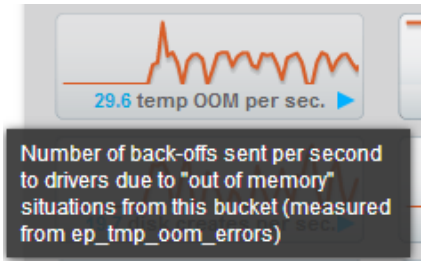
The command will not be able to put all 800 items in the bucket.

```
Failed to store item: [11] (Temporary failure received from server. Try again later) retry 10s
```

You may see this message in the GUI

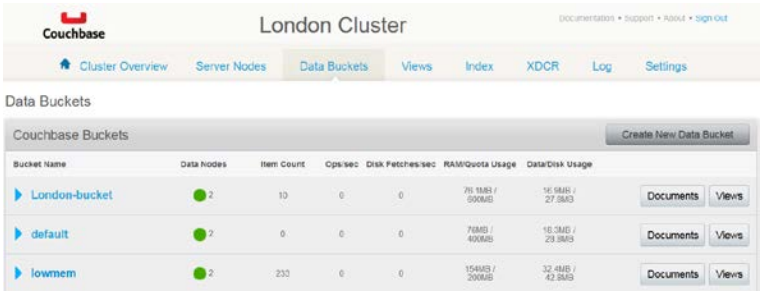


Note that in the performance metrics for the 'lowmem' bucket, in the "Summary" section, you will also find a bunch of temp OOM per second messages showing up (you can safely ignore these for our demo purposes, but should be very careful if you see this in a production environment!)



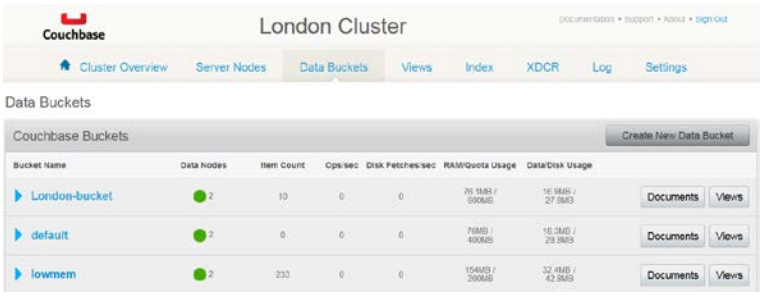
Once the command if finished, switch to the Web UI of the 2-node London cluster and click on Data Buckets at the top. You should see approximately 370-380 items in the lowmem bucket:

Note: If you only see 200-300 what could have caused this to occur?.

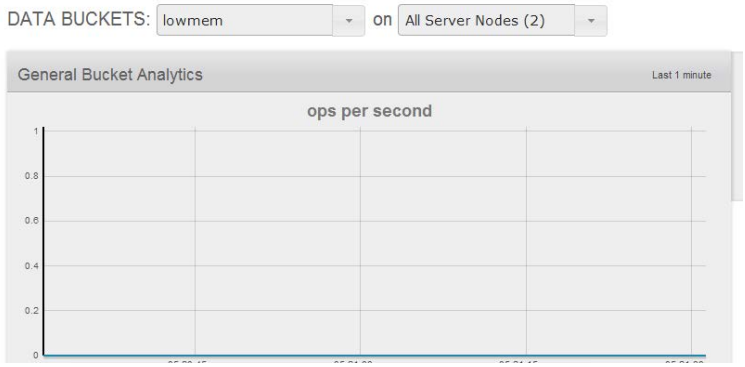


The most interesting metrics will be found in the performance graphs.

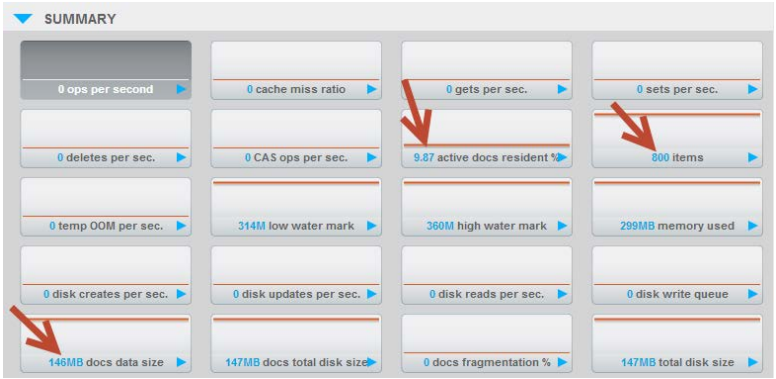
Click on the 'lowmem' bucket to load its metrics:



By the time, the ops per second for the lowmem bucket should be zero since pillowfight has finished running:



Scroll down to the SUMMARY section for 'lowmem' and notice that only about 10% of the 800 items we wrote fit in memory:

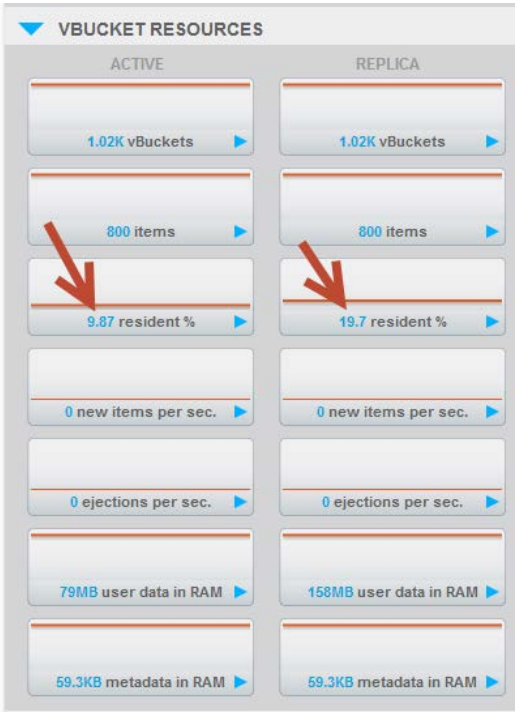


In row 2, column 4, notice that there are now 200-300 items (of approximately 800 sent in the command line syntax) in the bucket.

Also, notice in the summary graphs above in the 5th row, 1st column, the 'docs data size' is only 37MB. How would you explain this? We obviously just tried to write 800 MB of data to the 2-node cluster. (remember the pillowfight will not retry after a failed operation)

Remember that the data is compressed on disk, so the final compressed size is around 154 MB.

Next, expand the VBUCKET RESOURCES for the 'lowmem' bucket and notice that around 10% of the Active items are in memory, and around 20% of the replica items are in memory:



Note that in the phenomenon of seeing 20% of the replicas in RAM vs. 10% of the active items in RAM is an edge case scenario. The writes we did involved low percentages, low item counts and was not a realistic real-world workload. We also have performed zero reads against this data set so far. In the next section we will do a 100% read workload against this data set. In general, in the real world, you will see more of the active items in RAM than the replicas.

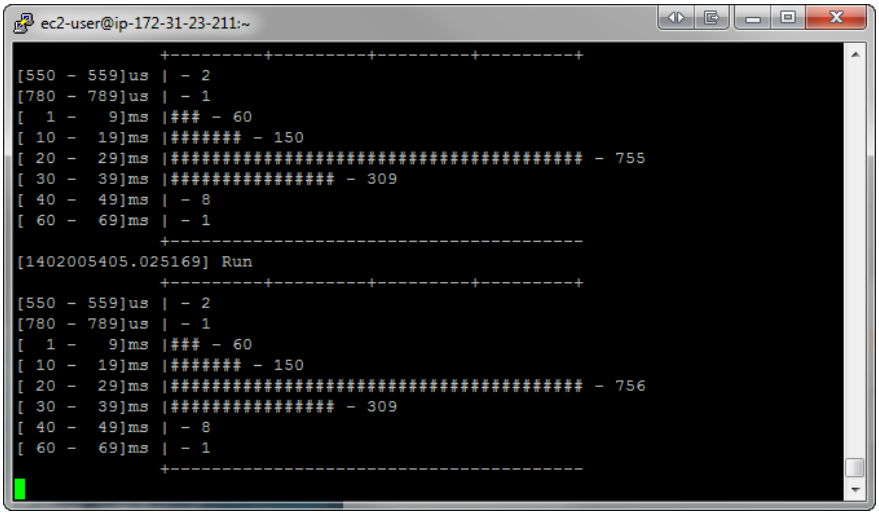
The next step is to use pillow fight to do 20,000 reads from the 'lowmem' bucket. Remember that since most of our data is on disk, the reads will be relatively slow (well, slower than

reading from RAM, anyway). Also, each item read is 1 MB, so it is somewhat large. The maximum size of an item that can be stored in Couchbase is 20 MB, but in practice most customers stored data in the KB range for each item.

Moving on, let's run pillow fight to do 790 reads over and over again (looped) and print histogram statistics about the read latency. (Note we are not reading 800 items, b/c in some cases pillowfight may write less than 800 keys into the cluster)

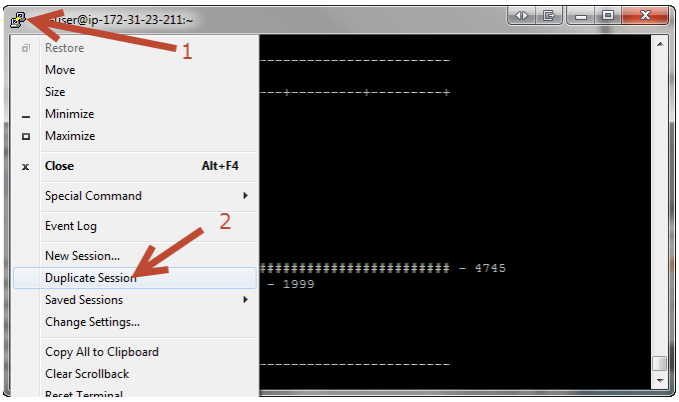
```
[ec2-user@ip-172-31-23-211 ~]$ cbc-pillowfight -U couchbase://ec2-54-172-130-44.compute-1.amazonaws.com/lowmem  --num-items=790 --set-pct=0 --timings -l
```

Once the command starts, you will see hundreds of graphs like this populating in the window showing the latency of the reads. Notice that most reads are between 20 – 29 ms b/c they are having to go to disk:

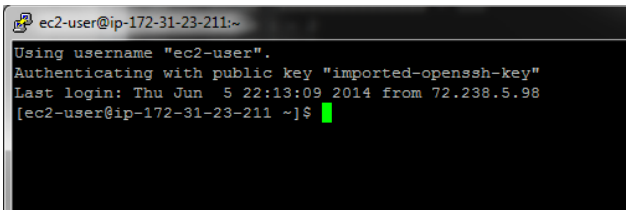


Keep in mind that our App Server is having to do a roundtrip network request to the 2 Couchbase nodes. Just the network roundtrip adds about 10 – 20 ms latency in Amazon's cloud (which has very variable network performance).

Test this out by opening a new PuTTY session for the App Server and sending some pings from the App Server to the 1st Node in the 2-node XDCR London cluster. (recall that this is 2-node cluster is not really in London, but in the us-east data center)



A new PuTTY window should open:



Run a ping command from the App Server to Node #1 in the 2-node London cluster:

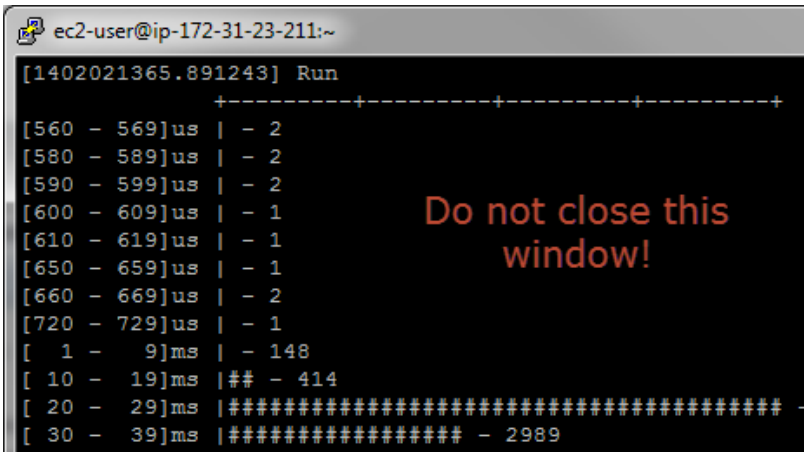
```
[ec2-user@ip-172-31-23-211 ~]$ ping ec2-54-86-94-157.compute-1.amazonaws.com
PING ec2-54-86-94-157.compute-1.amazonaws.com (172.31.41.210) 56(84) bytes of data.
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=1 ttl=64 time=0.501 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=2 ttl=64 time=16.1 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=3 ttl=64 time=0.503 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=4 ttl=64 time=12.7 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=5 ttl=64 time=22.8 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=6 ttl=64 time=3.12 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=7 ttl=64 time=16.1 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=8 ttl=64 time=2.16 ms
64 bytes from ip-172-31-41-210.ec2.internal (172.31.41.210): icmp_seq=9 ttl=64 time=16.1 ms
```

Hit <CTRL> + C to stop the ping command

Notice how variable the ping times are in the cloud.

You may now close the 2nd PuTTY window if you'd like.

However, keep the pillowfight window that is doing the reads open and running!



Switch to the Web UI for the 2-node London cluster and go to the performance metrics page for the 'lowmem' bucket across 2 nodes. Scroll down to the Summary graph and notice the following parameters:

In the graphs above:

- The cluster is handling about 35 gets per second

- The "% of active docs resident" has increased to around 65%. This is because the most recently read data is given priority to remain in memory over the replicas. You will see this explained further in the next screenshot.

- There are around 10 disk reads happening per second. This means that since there are 35 gets per second, 10 are going to disk, and 25 are served from memory. This makes sense. Since the 'active docs resident %' in memory is 65 % we are going to disk for about 30 – 35% of the reads (which is around 10 reads per second).

Next scroll down to the VBUCKET RESOURCES section:

Notice that:

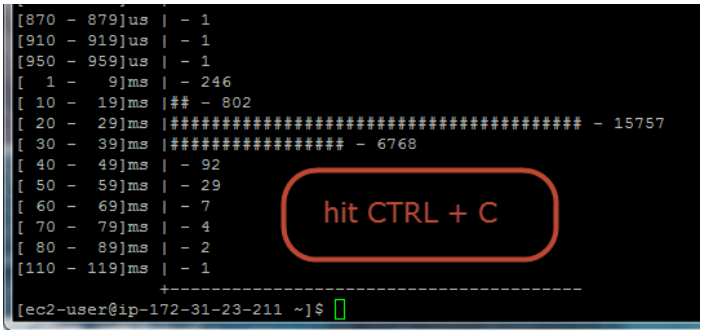
- around 60 – 70% of the active items are resident in memory

- There are about 60 ejections occurring per second. This is because we are reading all 790 keys in order and then reading the 790 keys in order again and repeating this process. So, data from disk is constantly being loaded into memory and to make room for this data, we're having to eject older items out of memory.

- Also in the 2nd column, notice that there is 0% of the replicas in memory. When the demand for RAM is high for the reads of the active data set, the replicas all get ejected out of memory.

Now that we are finished with this demonstration, let's stop pillowfight.

Switch back to the App Server (black VM) and hit <CTRL> + C to stop the reads from pillowfight:

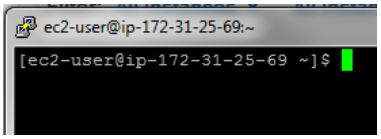


In the next section we'll use cbstats to display some metrics about the workload we ran in this section (800 writes, many reads from disk).

Performance Demo #3: Displaying metrics via cbstats:

As a Couchbase Server administrator, it is critical to understand the profile of your application traffic so that you can configure settings appropriately. cbstats is a tool that comes with Couchbase to see many metrics that are not available via the Web UI or the REST API. We will explore some of those metrics in this section.

Switch over to the App Server (black VM):



Let's get some statistics about the active vBuckets for the lowmem bucket that we created earlier in this lab. Remember that in the previous section we wrote about 800 items of size 1 MB to this bucket and then did A LOT of reads against it.
Keep in mind that all of the cbstats metrics are specific to the node that it is running against and to the bucket it is running against.

Run the following command using the public hostname of Node #1 in the 2-node London cluster:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-1.amazonaws.com:11210 all -b lowmem | egrep 'vb_active'
 vb_active_curr_items:             400
vb_active_eject:                  4274
vb_active_expired:                0
vb_active_ht_memory:              12845056
vb_active_itm_memory:             137398480
vb_active_meta_data_memory:       30248
vb_active_num:                    512
vb_active_num_non_resident:       267
vb_active_ops_create:             400
vb_active_ops_delete:             0
vb_active_ops_reject:             0
vb_active_ops_update:             0
vb_active_perc_mem_resident:      32
vb_active_queue_age:              0
vb_active_queue_drain:            400
vb_active_queue_fill:             400
vb_active_queue_memory:           0
vb_active_queue_pending:          0
vb_active_queue_size:             0
```

Here is an explanation of some of the above output. Keep in mind that some of the output is specific for only Node #1 in the 2-node XDCR/London cluster! You would have to run the command against Node #2's public hostname to get similar metrics for that node.

`vb_active_curr_items:` **This is the # of items currently on this node. So, in my specific setup, there are 400 items on Node #1 for the 'lowmem' bucket. This makes sense, since we wrote 800 items, we'd want the other node to have half of the data.**

`vb_active_eject:` **This refers to the # of times item values got ejected out of memory. As we were doing repeated reads of the 800 items in memory in the previous section, stale items kept getting ejected out of memory.**

`vb_active_itm_memory:` **Total item memory is displayed in bytes. For my specific setup, this translates to 131 MB. This is the amount of memory in active vBuckets specific to this node.**

`vb_active_num:` **This node is currently hosting 512 active vBuckets**

`vb_active_ops_create:` **This node saw 400 create operations**

The rest of the metrics in the above output are explained here:
http://docs.couchbase.com/admin/admin/CLI/cbstats-intro.html

Here are some more interesting metrics to pull out of the cbstats command. Run all of these commands against the 1st Node of the 2-node London/XDCR cluster:

How about the # of current items created in the lowmem bucket:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-
1.amazonaws.com:11210 all -b lowmem | egrep 'curr_items_tot'
curr_items_tot:                 800
```

Each document stored in the database has an optional expiration value (TTL, time to live). The default is for there to be no expiration, i.e. the information will be stored indefinitely. The expiration can be used for data that naturally has a limited life that you want to be automatically deleted from the entire database.

The expiration value is user-specified on a document basis at the point when the data is stored. The expiration can also be updated when the data is updated, or explicitly changed through the Couchbase protocol. The expiration time can either be specified as a relative time (for example, in 60 seconds), or absolute time (31st December 2012, 12:00pm).

Typical uses for an expiration value include web session data, where you want the actively stored information to be removed from the system if the user activity has stopped and not been explicitly deleted. The data will time out and be removed from the system, freeing up RAM and disk for more active data.

The expiration process in Couchbase Server uses the metadata in RAM to quickly scan for items that are expired and later remove them from disk. This process is known as the "expiry pager" and runs every 60 minutes by default.

Check if any items have been expired out of the 800 items:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-1.amazonaws.com:11210 all -b lowmem | egrep 'expired'
 ep_expired_access:                 0
 ep_expired_pager:                  0
 ep_item_flush_expired:             0
 vb_active_expired:                 0
 vb_pending_expired:                0
 vb_replica_expired:                0
```

No items have expired yet, which makes sense. There was no TTL set for any of the 800 items we wrote previously.

Warning: You will want to run the next 3 red commands within 60 seconds, so move quick!

Let's create a document using the 'cbc create' command and set the expiration time to 30 seconds.

```
[ec2-user@ip-172-31-23-211 ~]$ cbc create –U  couchbase://ec2-54-86-94-x.compute-1.amazonaws.com/lowmem -e 60 cbc_key<hit enter>
newdata<hit CTRL + D on the keyboard about 3 times to send an EOF character>
Stored "cbc_key" CAS:c00ad2a6172f0000
```

Then we'll immediately read the key in less than 60 seconds and make sure that it is definitely in the cluster.

```
[ec2-user@ip-172-31-23-211 ~]$ cbc cat -U couchbase://ec2-54-86-94-x.compute-1.amazonaws.com/lowmem  cbc_key
"cbc_key" Size:7 Flags:0 CAS:d69b7ad842020000
newdata
```

Also, quickly check the expiration metrics again:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-1.amazonaws.com:11210 all -b lowmem | egrep 'expired'
 ep_expired_access:                 0
 ep_expired_pager:                  0
 ep_item_flush_expired:             0
 vb_active_expired:                 0
 vb_pending_expired:                0
 vb_replica_expired:                0
```

Nothing has expired yet.

Now wait for the 60 second period to finish. Once you are confident that more than 60 seconds have passed, check the expiration metrics again:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-
1.amazonaws.com:11210 all -b lowmem | egrep 'expired'
 ep_expired_access:                0
 ep_expired_pager:                 0
 ep_item_flush_expired:            0
 vb_active_expired:                0
 vb_pending_expired:               0
 vb_replica_expired:               0
```

Why do you think the item has not expired yet?

The answer is that the expiry pager has not run yet. However there is another way to force a stale document to expire: try reading it.

Check if the document is still available with the cbc cat command:

```
[ec2-user@ip-172-31-23-211 ~]$ cbc cat –U couchbase://ec2-54-86-94-
x.compute-1.amazonaws.com -b lowmem  cbc_key
Failed to get "cbc_key": The key does not exist on the server
```

Check the expiration metrics again:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-157.compute-
1.amazonaws.com:11210 all -b lowmem | egrep 'expired'
 ep_expired_access:                0
 ep_expired_pager:                 0
 ep_item_flush_expired:            0
 vb_active_expired:                0
 vb_pending_expired:               0
 vb_replica_expired:               0
```

Why do you think the item has STILL not expired yet? Obviously the read was unsuccessful b/c the cluster realizes that the TTL has been reached and the item should not be provided to client applications any more.
Or perhaps in your environment the item does appear as expired?
The answer to this mystery (at least in my environment) is that cbstats' output is specific to the node you're running it against. If I run the same command against node #2 in the 2-node London cluster, I should see it as expired:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-96-x.compute-
1.amazonaws.com:11210 all -b lowmem | egrep 'expired'
 ep_expired_access:                1
 ep_expired_pager:                 0
 ep_item_flush_expired:            0
 vb_active_expired:                1
 vb_pending_expired:               0
 vb_replica_expired:               0
```

Note in the above output that the item was marked expired because it was accessed. Had the item pager run (which it does by default every hour) and marked it as expired, then the +1 would have occurred under the "ep_expired_pager" metric. The vb_active_expired metric is simply stating that 1 of the active items on this node has expired.

One of the most useful metrics to collect from cbstats is timing statistics. Timing stats provide histogram data from high resolution timers over various operations within the system.

From the App Server, run the following command against Node #1 of the 2-node London cluster:

```
[ec2-user@ip-172-31-23-211 ~]$ cbstats ec2-54-86-94-x.compute-1.amazonaws.com:11210 timings -b lowmem
 bg_wait (690 total)
    32us - 64us   : ( 66.96%) 462 ##############################
    64us - 128us  : ( 95.36%) 196 ############
    128us - 256us : ( 98.70%)  23 #
    256us - 512us : ( 99.13%)   3
    1ms - 2ms     : ( 99.42%)   2
    2ms - 4ms     : ( 99.71%)   2
    8ms - 16ms    : (100.00%)   2
    Avg           : (   77us)
 get_stats_cmd (3565 total)
    2us - 4us     : (  0.11%)    4
    8us - 16us    : (  0.14%)    1
    32us - 64us   : (  0.28%)    5
    64us - 128us  : (  0.79%)   18
    128us - 256us : (  3.81%)  108 #
    256us - 512us : ( 56.83%) 1890 ######################
    512us - 1ms   : ( 88.39%) 1125 #############
    1ms - 2ms     : ( 89.26%)   31
    2ms - 4ms     : ( 95.04%)  206 ##
    4ms - 8ms     : ( 96.16%)   40
    8ms - 16ms    : ( 98.12%)   70
    16ms - 32ms   : ( 99.55%)   51
    32ms - 65ms   : (100.00%)   16
    Avg           : (    1ms)
 disk_del (1 total)
    16us - 32us : (100.00%) 1 ##################################################
    Avg         : (   16us)
 disk_insert (800 total)
    16us - 32us   : (  0.12%)   1
    256us - 512us : ( 90.00%) 719 ########################################
    512us - 1ms   : ( 95.75%)  46 ##
    1ms - 2ms     : ( 96.75%)   8
    2ms - 4ms     : ( 97.75%)   8
    4ms - 8ms     : ( 98.50%)   6
    8ms - 16ms    : ( 99.38%)   7
    16ms - 32ms   : (100.00%)   5
    Avg           : (  495us)
 storage_age (801 total)
    0 - 1s        : ( 26.34%) 211 ############
    1s - 2s       : ( 57.80%) 252 ##############
    2s - 4s       : ( 74.78%) 136 #######
    4s - 7s       : ( 79.15%)  35 ##
    7s - 10s      : ( 80.27%)   9
    34s - 49s     : ( 81.27%)   8
    49s - 1m:09s  : (100.00%) 150 ########
    Avg           : (     5s)
 get_cmd (2747 total)
    2us - 4us     : ( 21.00%) 577 #########
    4us - 8us     : ( 53.00%) 879 ###############
    8us - 16us    : ( 72.81%) 544 #########
    16us - 32us   : ( 88.10%) 420 ######
    32us - 64us   : ( 98.69%) 291 ####
    64us - 128us  : ( 99.24%)  15
    128us - 256us : ( 99.34%)   3
    256us - 512us : ( 99.38%)   1
    1ms - 2ms     : ( 99.89%)  14
    2ms - 4ms     : ( 99.96%)   2
```

```
    4ms - 8ms     : (100.00%)   1
    Avg           : (   17us)
 tap_vb_reset (512 total)
    8us - 16us    : ( 16.99%)  87 #######
    16us - 32us   : ( 73.24%) 288 ########################
    32us - 64us   : ( 75.00%)   9
    64us - 128us  : ( 93.36%)  94 ########
    128us - 256us : ( 97.46%)  21 #
    256us - 512us : ( 98.63%)   6
    512us - 1ms   : ( 98.83%)   1
    1ms - 2ms     : ( 99.02%)   1
    2ms - 4ms     : ( 99.22%)   1
    4ms - 8ms     : ( 99.61%)   2
    8ms - 16ms    : (100.00%)   2
    Avg           : (   85us)
 disk_vbstate_snapshot (10505 total)
    1us - 2us     : (  0.02%)    2
    2ms - 4ms     : (  0.10%)    8
    4ms - 8ms     : ( 62.17%) 6521 ############################
    8ms - 16ms    : ( 85.19%) 2418 ##########
    16ms - 32ms   : ( 95.48%) 1081 ####
    32ms - 65ms   : ( 99.30%)  401 #
    65ms - 131ms  : ( 99.87%)   60
    131ms - 262ms : ( 99.92%)    6
    262ms - 524ms : ( 99.93%)    1
    1s - 2s       : ( 99.97%)    4
    2s - 4s       : ( 99.98%)    1
    16s - 33s     : ( 99.99%)    1
    33s - 1m:07s  : (100.00%)    1
    Avg           : (   13ms)
 store_cmd (3096 total)
    8us - 16us    : (  5.49%)  170 ##
    16us - 32us   : ( 12.02%)  202 ##
    32us - 64us   : ( 12.50%)   15
    64us - 128us  : ( 15.57%)   95 #
    128us - 256us : ( 93.22%) 2404 ##################################
    256us - 512us : ( 98.13%)  152 ##
    512us - 1ms   : ( 98.35%)    7
    1ms - 2ms     : ( 98.51%)    5
    2ms - 4ms     : ( 98.90%)   12
    4ms - 8ms     : ( 99.16%)    8
    8ms - 16ms    : ( 99.64%)   15
    16ms - 32ms   : ( 99.87%)    7
    32ms - 65ms   : (100.00%)    4
    Avg           : (  255us)
 tap_mutation (401 total)
    16us - 32us   : (  0.25%)   1
    128us - 256us : (  9.23%)  36 ####
    256us - 512us : ( 93.52%) 338 ########################################
    512us - 1ms   : ( 95.01%)   6
    1ms - 2ms     : ( 95.26%)   1
    2ms - 4ms     : ( 96.01%)   3
    4ms - 8ms     : ( 96.51%)   2
    8ms - 16ms    : ( 97.76%)   5
    16ms - 32ms   : (100.00%)   9 #
    Avg           : (  743us)
 bg_tap_load (1442 total)
    512us - 1ms : (  2.36%)  34 #
    1ms - 2ms   : ( 64.84%) 901 ##############################
    2ms - 4ms   : ( 76.63%) 170 #####
    4ms - 8ms   : ( 82.80%)  89 ##
    8ms - 16ms  : ( 95.21%) 179 #####
    16ms - 32ms : ( 99.45%)  61 #
    32ms - 65ms : (100.00%)   8
    Avg         : (    3ms)
 bg_tap_wait (1442 total)
    16us - 32us   : (  0.28%)   4
    32us - 64us   : (  1.32%)  15
    64us - 128us  : (  2.01%)  10
```

```
    128us - 256us : (  6.66%)  67 ##
    256us - 512us : (  8.74%)  30
    512us - 1ms   : ( 10.06%)  19
    1ms - 2ms     : ( 16.09%)  87 ##
    2ms - 4ms     : ( 24.06%) 115 ###
    4ms - 8ms     : ( 31.41%) 106 ###
    8ms - 16ms    : ( 39.53%) 117 ###
    16ms - 32ms   : ( 49.93%) 150 ####
    32ms - 65ms   : ( 62.07%) 175 #####
    65ms - 131ms  : ( 74.41%) 178 #####
    131ms - 262ms : ( 89.18%) 213 ######
    262ms - 524ms : ( 99.93%) 155 ####
    524ms - 1s    : (100.00%)   1
    Avg           : (   62ms)
 notify_io (1592 total)
    1us - 2us     : (  0.75%)  12
    2us - 4us     : (  0.82%)   1
    4us - 8us     : (  1.70%)  14
    8us - 16us    : ( 29.65%) 445 ############
    16us - 32us   : ( 54.52%) 396 ###########
    32us - 64us   : ( 55.21%)  11
    64us - 128us  : ( 56.34%)  18
    128us - 256us : ( 56.91%)   9
    256us - 512us : ( 87.88%) 493 #############
    512us - 1ms   : ( 91.96%)  65 #
    1ms - 2ms     : ( 93.34%)  22
    2ms - 4ms     : ( 96.36%)  48 #
    4ms - 8ms     : ( 97.36%)  16
    8ms - 16ms    : ( 99.25%)  30
    16ms - 32ms   : ( 99.87%)  10
    32ms - 65ms   : (100.00%)   2
    Avg           : (  523us)
 set_vb_cmd (1024 total)
    16us - 32us   : ( 77.25%) 791 ###################################
    32us - 64us   : ( 97.56%) 208 #########
    64us - 128us  : ( 98.83%)  13
    128us - 256us : ( 98.93%)   1
    256us - 512us : ( 99.32%)   4
    512us - 1ms   : ( 99.41%)   1
    2ms - 4ms     : ( 99.71%)   3
    8ms - 16ms    : ( 99.80%)   1
    16ms - 32ms   : ( 99.90%)   1
    32ms - 65ms   : (100.00%)   1
    Avg           : (   83us)
 disk_commit (14438 total)
    0 - 1s  : ( 99.99%) 14437 ##################################################
    1s - 2s : (100.00%)     1
    Avg     : (     1s)
 item_alloc_sizes (3096 total)
    1MB - 2MB : (100.00%) 3096 ##################################################
    Avg       : (    1MB)
 batch_read (690 total)
    1ms - 2ms     : ( 64.06%) 442 ##############################
    2ms - 4ms     : ( 85.36%) 147 #########
    4ms - 8ms     : ( 91.45%)  42 ##
    8ms - 16ms    : ( 95.22%)  26 #
    16ms - 32ms   : ( 98.12%)  20 #
    32ms - 65ms   : ( 99.86%)  12
    65ms - 131ms  : (100.00%)   1
    Avg           : (    2ms)
 bg_load (690 total)
    512us - 1ms   : (  0.14%)   1
    1ms - 2ms     : ( 84.06%) 579 ########################################
    2ms - 4ms     : ( 90.29%)  43 ##
    4ms - 8ms     : ( 93.62%)  23 #
    8ms - 16ms    : ( 95.51%)  13
    16ms - 32ms   : ( 98.99%)  24 #
    32ms - 65ms   : ( 99.86%)   6
    65ms - 131ms  : (100.00%)   1
```

```
    Avg          : (    2ms)
```

I've highlighted some of the more important metrics above and an explanation for them follows:

`disk_del`: This will be 1 because so far we have only deleted 1 item (which expired on access) from the bucket. This metric is specifically a histogram of time spent waiting for disk to delete an item, which for our specific delete was `16us - 32us` microseconds.

The get_cmd below is time spent servicing get requests. The # 2747 total below means that we read against this bucket 2747 times from pillow fight (and a few times via the cbc cat command). Note that most gets occurred in under 64 microseconds.

```
get_cmd (2747 total)
    2us - 4us      : ( 21.00%) 577 #########
    4us - 8us      : ( 53.00%) 879 ##############
    8us - 16us     : ( 72.81%) 544 ########
    16us - 32us    : ( 88.10%) 420 ######
    32us - 64us    : ( 98.69%) 291 ####
    64us - 128us   : ( 99.24%)  15
    128us - 256us  : ( 99.34%)   3
    256us - 512us  : ( 99.38%)   1
    1ms - 2ms      : ( 99.89%)  14
    2ms - 4ms      : ( 99.96%)   2
    4ms - 8ms      : (100.00%)   1
```

The item_alloc_sizes shows that of the 3,096 items in the bucket, 100% of them are between 1 MB and 2 MB in size:

```
item_alloc_sizes (3096 total)
    1MB - 2MB : (100.00%) 3096
################################################
    Avg       : (    1MB)
```

Finally, the bg_load metric shows that 690 times there was a need for a background fetch that required reading from disk. Almost all of the time, this finished in under 2 ms.

```
bg_load (690 total)
    512us - 1ms   : (  0.14%)   1
    1ms - 2ms     : ( 84.06%) 579
########################################
    2ms - 4ms     : ( 90.29%)  43 ##
    4ms - 8ms     : ( 93.62%)  23 #
    8ms - 16ms    : ( 95.51%)  13
```

```
    16ms - 32ms  : ( 98.99%)  24 #
    32ms - 65ms  : ( 99.86%)   6
    65ms - 131ms : (100.00%)   1
    Avg          : (    2ms)
```

More details about timing statistics can be found here:
http://docs.couchbase.com/admin/admin/CLI/CBstats/cbstats-timing.html

Compaction:

All inserts, updates and deletes in Couchbase are appended to a data file on disk. There are 1024 vBuckets in each bucket and therefore 1024 data files under each bucket. Couchbase's append-only engine can eventually lead to gaps within the data file (particularly when data is deleted) which can be reclaimed using a process called compaction.

The index files that are created each time a view is built are also written in a sequential format. Updated index information is appended to the file as updates to the stored information is indexed.

Couchbase compacts views and data files. For database compaction, a new file is created into which the active (non-stale) information is written. Meanwhile, the existing database files stay in place and continue to be used for storing information and updating the index data. This process ensures that the database continues to be available while compaction takes place. Once compaction is completed, the old database is disabled and saved. Then any incoming updates continue in the newly created database files. The old database is then deleted from the system.

View compaction occurs in the same way. Couchbase creates a new index file for each active design document. Then Couchbase takes this new index file and writes active index information into it. Old index files are handled in the same way old data files are handled during compaction. Once compaction is complete, the old index files are deleted from the system.

Compaction takes place as a background process while Couchbase Server is running. You do not need to shutdown or pause your database operation, and clients can continue to access and submit requests while the database is running. While compaction takes place in the background, you need to pay attention to certain factors.

Make sure you perform compaction…

- …on every server
- …during off-peak hours (The compaction process is both disk and CPU intensive)
- …with adequate disk space

Regarding the last point, because compaction occurs by creating new files and updating the information, you may need as much as twice the disk space of your current database and index files for compaction to take place.

However, it is important to keep in mind that the exact amount of the disk space required depends on the level of fragmentation, the amount of dead data and the activity of the database, as changes during compaction will also need to be written to the updated data files.

Before compaction takes place, the disk space is checked. If the amount of available disk space is less than twice the current database size, the compaction process does not take place and a warning is issued in the log.
Couchbase Server incorporates an automated compaction mechanism that can compact both data files and the view index files, based on triggers that measure the current fragmentation level within the database and view index data files.

Actually there are 3 ways that auto-compaction can get triggered:

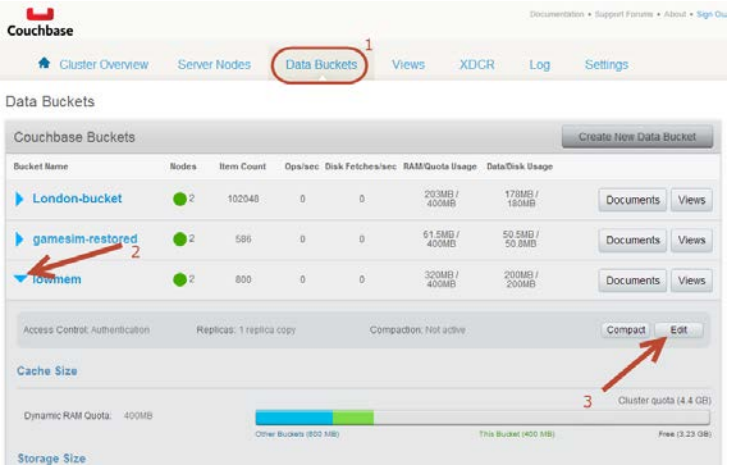
- Database fragmentation. This is the primary setting. If you set this figure to 10%, then the moment the data files are 10% fragmented, auto-compaction will be triggered (unless you have time-limited auto-compaction)
- View fragmentation. This specifies the percentage of fragmentation within all view index files at which compaction will be triggered
- Time period. To prevent auto compaction taking place when your database is in heavy use, you can configure a time during which compaction is allowed. This is expressed as the hour and minute combination between which compaction occurs. For example, you could configure compaction to take place between 01:00 and 06:00.

By default, compaction operates sequentially, executing first on the database and then the Views if both are configured for auto-compaction. If you enable parallel compaction, both the databases and the views can be compacted at the same time. This requires more CPU and database activity for both to be processed simultaneously, but if you have CPU cores and disk I/O (for example, if the database and view index information is stored on different physical disk devices), the two can complete in a shorter time.

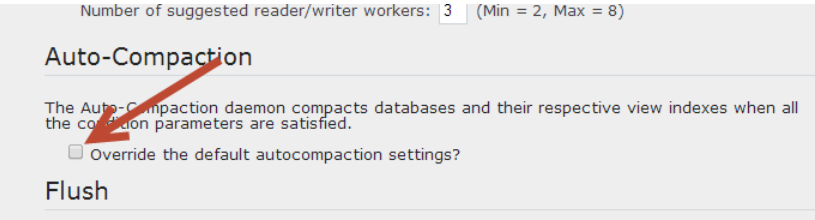
Another setting for compaction is the metadata purge interval. You can remove tombstones for expired and deleted items as part of the auto-compaction process. Tombstones are records containing the key and metadata for deleted and expired items and are used for eventually consistency between clusters and for views.

Enough discussion. Let's take a look at the compaction settings for the lowmem bucket.

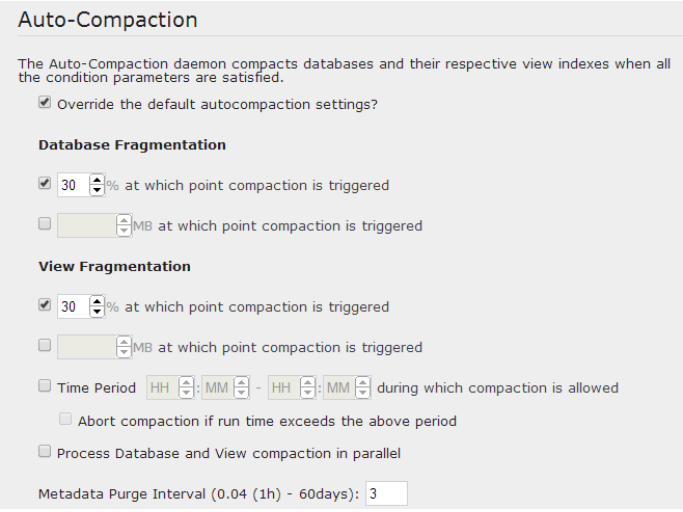
Switch to the browser tab for the 2-node London cluster and click on 'Data Buckets' at the top. Then expand the settings for the 'lowmem' bucket by clicking the blue arrow next to it. Finally, click Edit:



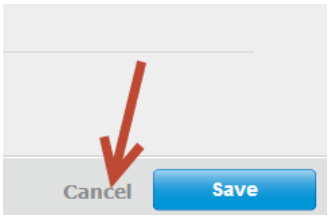
In the pop-up, scroll down till you see Auto-Compaction and place a check to override the default settings:



Spend a minute familiarizing yourself with the different settings here, but don't change anything for now.

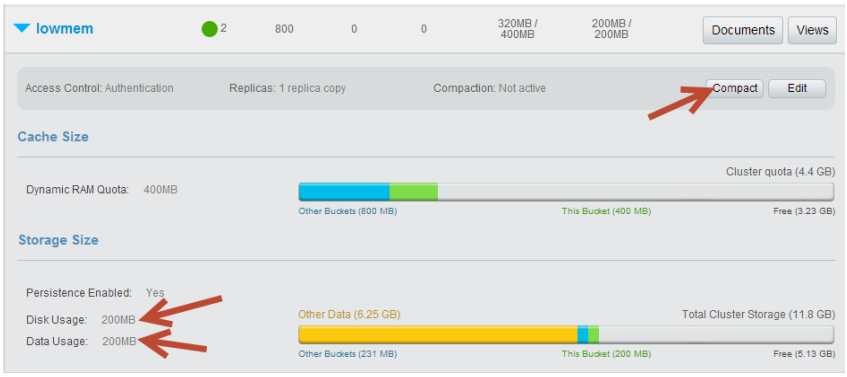


When you are finished, click on Cancel:

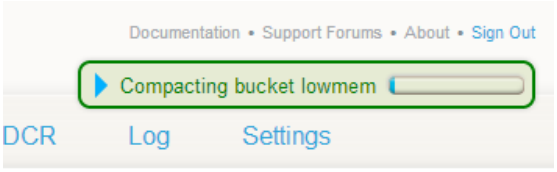


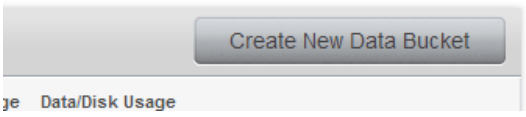
You should now be back at the Data Buckets page. Notice that the 'lowmem' bucket is using around 200 MB of disk space and has about 200 MB of actual data. This means that the bucket is pretty much 0% fragmented (the only item that we deleted was the cbc_key that expired with the 60 second TTL hit). If the data usage was 200 MB, but the disk usage was 300 MB, that means that the bucket is VERY fragmented with about 50% of extra data that it doesn't need to maintain (like deletes, expired items and older updates).

Even though this bucket does not need compaction run, let's try to run it anyway to see what would happen. Click on Compact for the 'lowmem' bucket:



Once compaction starts, you'll see a green hovering rectangle in the GUI notifying you of the compaction process.



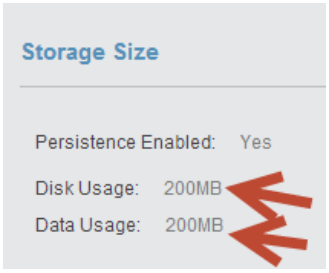


Compaction will take about 1 – 2 minutes to run.

In the web UI, you can also see what percentage of compaction is completed:

Once the green rectangle has disappeared, continue with the rest of this lab…

Look at the Storage Size again and you will not notice a difference. This is b/c the data files were pretty much unfragmented to begin with.



This concludes lab #8.