Automated Sharded MongoDB Deployment and Benchmarking for Big Data Analysis

MARK MCCOMBE[1,*]

[1]*School of Informatics and Computing, Bloomington, IN 47408, U.S.A.*
[*]*Corresponding authors: laszewski@gmail.com*

Using Python, Ansible, and Cloudmesh Client scripts, a fully automated process is created for deploying a configurable MongoDB sharded cluster on Chameleon, FutureSystems, Jetstream, and (possibly) Amazon Web Services (AWS) cloud computing environments. A user runs one program, which configures and deploys the environment based on input parameters for Config Servers, Mongos Instances, Shards, and a Replication Factor. Additionally, the automated process will run multiple benchmarking tests for each deployment, capturing statistics in a file as input for python (STILL EVALUATING OPTIONS) visualization programs, the results of which are displayed in this report. As background, technologies and concepts key to the deployment and benchmarking, such as MongoDB, Python, Ansible, Cloudmesh Client, and Openstack are examined.

Keywords: MongoDB, Cloud Computing, Ansible, Python, Cloudmesh Client, Openstack, I524

https://github.com/cloudmesh/sp17-i524/tree/master/project/S17-IO-3012/report/report.pdf

1. INTRODUCTION

As the final project for I524, Big Data Software and Projects, Spring 2017, a Python program invoking Ansible playbooks has been created to fully automate a configurable deployment of a MongoDB sharded cluster on various clouds. Chameleon Cloud, FutureSystems, and Jetstream are the currently supported clouds. The scripts have been developed and tested on an Ubuntu 16.04 LTS (Xenial Xerus) Virtual Machine running in Virtual Box. In addition to Ansible, Cloudmesh Client is used for key interaction between the client and cloud environment. Options can be specified for deployment cloud, replication degree of the Config Server servers, number of Mongos instances, number of Data Shards, and the degree of replication within the Shards.

As part of the deployment, automated benchmarking tests are also run. Tests were performed with various sharding and replication configurations to assess their impact on performance. Performance results are captured and graphed using Python's matplotlib, the results of which are displayed and analyzed in this report.

2. EXECUTION PLAN

A rough execution plan is shown below. These steps are not necessarily sequential so many will be worked on concurrently. All functionality mentioned may not be added to final project and additional features may be added during development.

1. Convert my existing MongoDB deployment ksh prototype to Python cmd and Ansible

2. Test and utilize cloudmesh clusters

3. Test new version on Chameleon Cloud

4. Test new version on FutureSystem

5. Port to Jetstream and test

6. Depending on Cloudmesh functionality available, xplore porting to AWS. May do some testing of this early on to allow for building in to program since non OpenStack may work differently.

7. Explore adding x.509 security option to security option to existing keyfiles

8. Add Map/Reduce test to existing mongoimport and find benchmarking tests

9. Consider rewriting visualization originally done with python using another language or tool

10. Complete paper

2.1. Update 3/26/2016

1. Project approved by Professor based on proposal and execution plan.

2. Successfully tested Chameleon and FutureSystems cloud and cloudmesh access and commands.

3. Tested JetStream access, but unable to connect via cloudmesh. Question to Professor on Piazza.

4. Completed all Ansible lessons/assignements as preparation for project work. Beginning to work on incorporating to project.

5. Completed cloudmesh cluster tutorial. Beginning to work on incorporating to project.

6. Made modifitions to report.

3. INFRASTRUCTURE

Will make major modifications

Two clouds were selected for deployment: Chameleon Cloud and Futuresystems. In our automated deployment and benchmarking process, the -d option determines which cloud the deployment will be run on.

3.1. OpenStack

Both Chameleon Cloud and FutureSystems use OpenStack. OpenStack is a free, open source cloud computing platform, primarily deployed as IaaS. [1] Openstack was created in 2010 as joint project between NASA and Rackspace that is currently managed by the OpenStack Foundation. [1] Open Stack is open source software released under the Apache 2.0 license. [2]

Open Stack has various components, also known by code names [1]. Examples of Openstack components (and code names) are Compute (Nova), Networking (Neutron), Block Storage (Cinder), Identity (Keystone), Image (Glance), Object Storage (Swift), Dashboard (Horizon), Orchestration (Heat), Workflow (Mistral), Telemetry (Ceilometer), OpenStack Telemetry (Ceilometer), Database (Trove), Elastic Map Reduce (Sahara), Bare Metal (Ironic), Messaging (Zaqar), Shared File System (Manila), DNS (Designate), Search (Searchlight), and Key Manager (Barbican) [1].

3.2. Chameleon Cloud

Chameleon is funded by the National Science Foundation and provides computing resources to the open research community. The Chameleon testbed is hosted at the Texas Advanced Computing Center and the University of Chicago. Chameleon provides resources to facilitate research and development in areas such as Infrastructure as a Service, Platform as a Service, and Software as a Service. Chameleon provides both an OpenStack Cloud and Bare Metal High Performance Computing Resouces. [3]

If *chameleon* is specified for the -d parameter, the automated deployment and benchmarking process will be run on Chameleon Cloud.

3.3. FutureSystems

FutureSystems is a computing environment run by Indiana University that supports educational and research activities. [4] FutureSystems is directed by Geoffrey C. Fox and Gregor von Laszewski, both of Indiana University. [5] For our deployment,

we utilized the OpenStack Kilo Cloud, running on the India machine.

If *kilo* is specified for the -d parameter, the automated deployment and benchmarking process will be run on Kilo Cloud.

3.4. Jetstream

3.5. Amazon Web Services

3.6. Cloud Hardware Comparison

Will make major modifications

Figure 1 shows a comparison of several key computing resources on Chameleon, FutureSystems, and Jetstream cloud environments.

Need to add Jetstream info. https://jetstream-cloud.org/technology.php

	FutureSystems	Chameleon	Jetstream
CPU	Intel Xeon E5-2670 v3	Intel Xeon X5550	Dual Intel E-2680v3 "Haswell"
CPU cores	1024	1008	7680
CPU speed	2.66GHz	2.3GHz	2.5GHz
RAM	3072GB	5376GB	placeholder
RAM speed	1333Mhz	unknown	unknown
switches	Juniper/Dell EX Force 10	Dell S6000	
storage	335TB	1.5PB	2 TB

Fig. 1. Cloud Hardware Specification Comparison [6] [7]

MAYBE CHANGE THIS TO PER VM OBVIOUSLY REFORMAT, probably flip columns and rows

4. ANSIBLE

5. PYTHON/CMD

6. CLOUDMESH CLIENT

The Cloudmesh Client toolkit is an open source client interface that standardizes access to various clouds, clusters, and workstations. [8] Cloudmesh Client is a python based application developed by Gregor von Laszewski and others at Indiana University.

In the deployment, Cloudmesh Client is used to handle most interaction with the Virtual Machines in the clouds. The following tasks are performed via Cloudmesh Client:

1. Uploading a Public Key - Cloudmesh's key add and upload commands simplify key management.

2. Uploading Security Rules - Cloudmesh's secgroup commands allow new security rules to be added and uploaded to the cloud.

3. Booting Virtual Machines - Cloudmesh's vm boot command allow easy creation of virtual machines.

4. IP Address Management - Cloudmesh's functionality to assign and capture Floating IP Addresses is utilized

5. Deleting Virtual Machines - Cloudmesh simplifies deletion of VMs created in the deployment.

Cloudmesh Client simplifies and standardized interaction with the cloud for these tasks. This allows us to more easily port the deployment to additional clouds. Furthermore, by

encapsulating the logic necessary to perform these tasks we are shielded from changes in interfaces made by individual clouds.

More details of the use Cloudmesh Client in the automated deployment are included in Appendix B.

7. MONGODB

MongoDB is a popular open source, document oriented noSQL database. It stores documents in JSON-like schema-less formats called collections. [9] DBEngines ranks MongoDB as the most popular noSQL store and as the fifth most popular Database Management System overall. [10]

7.1. Architecture

A sharded cluster in MongoDB has three main components, all of which will be implemented in our deployment:

- Config Servers - hold configurations setting and metadata

- Mongos - a query routing interface between applications and the cluster
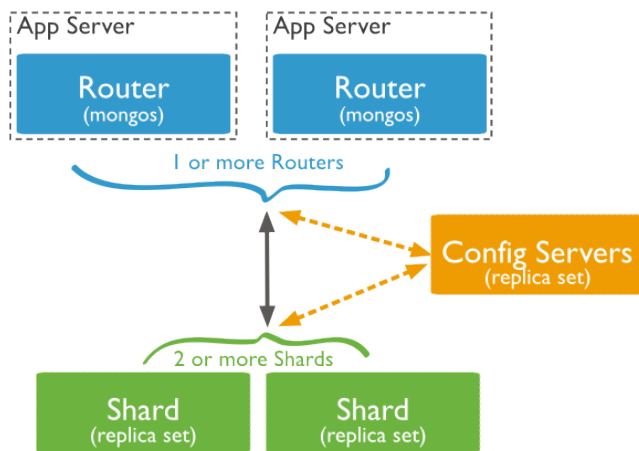
- Shards - subsets of data

Fig. 2. Sharded MongoDB Architecture [11]

Figure 2 depicts a sharded MongoDB environment with two Mongos instances and two data Shards. The replica sets shown for both Config Servers and Shards may have any number of replicas within the set.

7.2. Config Servers

Config Servers stored metadata for sharded MongoDB clusters. This metadata includes information about the state and structure of the data and components of the sharded cluster. [12]

Config Servers also contain authentication information. For example, information about the keyfiles used for internal authentication between the nodes (described in detail in the Security subsection that follow) is stored in the Config Servers. [12]

In production deployments, it is recommended for Config Servers to be deployed in 3 member replica sets. [11] The rationale behind a 3 member set is discussed in more detail in the Replication subsection that follows.

In our deployment and benchmarking automation, the degree of replication in the Config Server Replica Set is controlled

by the -c parameter. For example, specifying 1 for -c will create a Replica Set with 1 Config Servers, specifying 1 for -c will create a Replica Set with 2 Config Server, and so on.

7.3. Mongos Routers

Mongos is a query routing service used in sharded MongoDB configurations. Queries from applications go through Mongos, which locates the data in the sharded cluster. The Mongos instances accomplish this by reading and caching data from the Config Servers. [13]

For applications with high performance or availability requirements multiple Mongos instances may be ideal. In a high volume application, spreading the routing load over multiple Mongos instances can benefit performance. Additionally, multiple Mongos instances may increase availability in the case where a Mongos instance fails. [12]

In our deployment and benchmarking automation, the number of Mongos instances created is controlled by the -m parameter. For example, specifying 1 for -m will create 1 Mongos instance, specifying 2 for -m will create 2 Mongos instances, and so on.

7.4. Shards

Sharding, or distributing data across machines, is used by MongoDB to support large data sets and provide high throughput. [14]. Our deployment and benchmarking will test the performance of various numbers of shards, measuring the performance improvements associated with sharding in MongoDB.

Documents are distributed among the shards using a shard key. A sharded collection must have one, and only one, shard key which must have a supporting index. [14] Since the shard key is critical to performance and efficiency, particular care must be given to shard key selection. [15] In our performance testing a key was chosen that would distribute data relatively evenly across the shards, but was not used in retrieving the data as the more costly retrieval of not using an index provided a better test case.

In our deployment and benchmarking automation, Sharding is controlled by the -s parameter. For example, specifying 1 for -s will cause 1 shard to be created, specifying 2 will cause 2 shards to be created, and so on.

7.5. Replication

In databases, replication provides data redundancy leading to greater availability and fault tolerance. Replication in MongoDB is achieved via Replica Sets. [16] Replica Sets were implemented in our deployment for both Config Servers and Shards.

Two key benefits provided by replication are redundancy and fault tolerance.

- Redundancy - Each Replica in a Replica Set provides another copy of data. Higher redundancy means more nodes can be lost without data being lost.

- Fault Tolerance - Higher numbers of Replicas in a set increase fault tolerance, which leads to increased availability. As a general rule, a set will be able to tolerate faults while the majority of its nodes are still available

As shown in Figure 3, odd numbers of members in a replica set are a better choice for fault tolerance. For example, both a 3 and 4 replace set can only tolerate one member failing while maintaining availability. This is because a majority of the members must be available to maintain availability. In a 3 replica

Replica Members	Majority Needed	Fault Tolerance
2	1	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3

Fig. 3. Fault Tolerance by Replica Set Size
[17]

set the majority is 2, so it can tolerate 1 member failing. In a 4 replica set, the majority is 3, so it can still only tolerate 1 member failing. Increases in fault tolerance only occur when the next odd numbered member of a replica set is added. [17]

For production systems, a standard deployment is a 3 Replica Set. [17]. A 3 replica set provides 3 copies of the data for redundancy and fault tolerance if 1 member of the set were to fail. In a situation where availability was of higher concern, a 5 replica set would provide 4 copies of the data for redundancy and fault tolerance if 2 members of the set were to fail.

In our automated deployment and benchmarking process, the degree of replication for Shards is controlled by the -r option. For example, specifying 1 for -r will create a replica set per shard with only 1 copy of data (essentially no replication, although technically we create a 1 member replica set), specifying 2 will cause a replica set of 2 to be created, and so on.

7.6. Security

There are two levels of security to consider in a sharded MongoDB deployment: internal and external authentication.

In our deployment the various MongoDB components (config servers, mongos instances, shards, and replicas) all reside on separate Virtual Machines. These machines must be able to communicate with each other. Two steps were necessary to enable this internal authentication. First, the ports (27017, 27018, 27019, 28017) used by MongoDB needed to be opened for communication. This was accomplished by adding appropriate security group rules to the clouds through Cloudmesh client. Second, MongoDB requires the internal authentication to be done by either keyfiles or x.509 certificates. [18] In our deployment, authentication is done by keyfiles. CONSIDERING ADDING X.509 SECURITY OPTION

For external authentication, the three users were created.

- *admin* - The user *admin* is created with the role of *userAdminAnyDatabase*. This user performs administrative functions such as creating the other users.

- *cluster_admin_user* - The user *cluster_admin_user* is created with the role *clusterAdmin*. This user performs sharding functions such as sharding the collection and checking its data distribution.

- *user1* - This is a standard user with readWrite permissions. This user performs the benchmarking tests and other functions not requiring administrative privileges.

8. DEPLOYMENT

Will make major modifications
The automated process will fully deploy a sharded MongoDB environment with the number of Config Servers, Mongos Instances, Shards, and degree of Replication specified as input parameters. It will also automatically created a sharded collection, run benchmarking tests agains it, and capture the output in CSV files as input for python visualization programs.

8.1. Deployment Process

The deployment consists of Ansible playbooks and Cloudmesh Client scripts and commands. It is run on the user's local Ubuntu 16.04 instance and executes commands both locally and on virtual machines in the cloud over ssh. A high level overview of the process is provided here. An inventory and overview of the code used to automate the deployment process is provided in Appendix B.

The automated deployment is run by executing the top level script, mongo_deploy.sh [19] [20] The script takes several parameters, described below. The script will fully deploy the environment as specified.

THIS WILL DRASTICALLY CHANGE BUT LEAVING AS PLACEHOLDER TO MAKE SURE EVERYTHING IS INCLUDED AFTER CONVERSION TO ANSIBLE

- -d - the Deployment Cloud (*chameleon* or *kilo*)

- -c - a number, the size of Config Server Replica Set

- -s - number of Shards

- -r - a number, size of Shard Replica Set

After validating the input parameters the deployment is run as follows.

Security

1. Add and upload the user's public key (this functionality is currently disabled to prevent throwing an error as it is assumed the user has already done this).

2. Add security rules to the user's default security profile port opening ports 27107, 27018, 27019, and 28017 for communication.

3. Upload the security group profile.

4. Create a new random keyfile (described in more detail in the security section) for each deployment.

Config Servers

1. Boot the appropriate number of Virtual Machines based on the -c parameter.

2. Check if there are available Floating IP Addresses in the pool. If not, create a new one.

3. Assign a Floating IP to each Virtual Machine and record it in a file (the Floating IP is needed for communication between the servers and ssh from the client to the cloud).

4. Modify /etc/hosts to fix a warning message generated when running as sudo

5. Install MongoDB software on all servers.

6. Transfer the mongo.key file to the Virtual Machine using Secure Copy (scp)

7. Transfer a configuration file for the Config Server to the Virtual Machine using Secure Copy (scp)

8. Over ssh, stop the running mongod process on each server and start a new mongod processes using the configuration file sent in the previous step.

9. Over ssh, on one Config Server, run an initiate command including the IP addresses of the others to start the Replica Set.

Mongos Instances

1. Boot the appropriate number of Virtual Machines based on the -m parameter.

2. Check if there are available Floating IP Addresses in the pool. If not, create a new one.

3. Assign a Floating IP to each Virtual Machine and record it in a file (the Floating IP is needed for communication between the servers and ssh from the client to the cloud).

4. Modify /etc/hosts to fix a warning message generated when running as sudo

5. Install MongoDB software on all servers.

6. Transfer the mongo.key file to the Virtual Machine using Secure Copy (scp)

7. Transfer the configuration file for the mongos instance to each Virtual Machine using scp. Prior to transferring, update the template configuration file to include the IP Address of the Configuration Server created earlier.

8. Over ssh, stop the running mongod process on each server and start a new mongos processes using the configuration file sent in the previous step.

9. Create several users (detailed in security section)

Shards

1. For each -r Replica Set, boot the appropriate number of Virtual Machines based on the -s parameter.

2. Check if there are available Floating IP Addresses in the pool. If not, create a new one.

3. Assign a Floating IP to each Virtual Machine and record it in a file (the Floating IP is needed for communication between the servers and ssh from the client to the cloud).

4. Modify /etc/hosts to fix a warning message generated when running as sudo

5. Install MongoDB software on all servers.

6. Transfer the mongo.key file to the Virtual Machine using Secure Copy (scp)

7. Transfer the configuration file for the Shard to each Virtual Machine using scp. Prior to transferring, update the template configuration file to include the correct replica set name.

8. Over ssh, stop the running mongod process on each server and start a new mongod processes using the configuration file sent in the previous step.

9. Over ssh, on one shard in the replica set, run an initiate command including the IP addresses of the other shards in the set to start the Replica Set.

10. On a mongos instance, run a command to add the shard replica set.

After deployment and benchmarking (described in detail in a later section) are complete, all Virtual Machines that have been created in the current deployment are deleted.

8.2. Deployment Timing

Will make major modifications

The configuration parameters and total deployment time is captured in a file for each deployment (benchmarking timings are later added to this file as well). Total run time for a few interesting configurations are shown in Figure 4.

Deployment A shows a simple deployment with only one of each component being created. This deployment may only be suitable for a development or test environment. Deployment A takes 517 seconds.

Deployment B shows a more complex deployment with production like replication factors for Config Servers and Shards and an additional Mongos instance. This deployment may be suitable for a production environment as it has greater fault tolerance and redundancy. Deployment B takes 2533 seconds.

Deployment C shows a deployment focused on high performance. It has a high number of shards, 9, but no fault tolerance or redundancy. The deployment may be suitable where performance needs are high and availability is less critical. Deployment C takes 1968 seconds.

More detailed run times for these deployments are shown in Appendix B, along with benchmarking results.

	Config Servers	Mongos	Shards	Replicas	Time in
	-c	-m	-s	-r	Seconds
A	1	1	1	1	*517*
B	3	2	3	3	*2533*
C	1	1	9	1	*1968*

Fig. 4. Deployment Times on Chameleon Cloud in Seconds

The total number of virtual machines is highly correlated with deployment time as booting the machines and installing the software, tasks that occur for all nodes, take the most time. he additional steps to configure Config Servers, Mongos Instances, Replicas, and Shards run in relatively similar times, so the specific type of component created has little impact on the deployment time. For example, holding all other deployment variables at 1, a deployment with 5 Config Servers took 1258 seconds, one with 5 Mongos Instances took 1253 seconds, one with 5 Shards took 1292 seconds, and one with a 5 Shard Replica set took 1277 seconds.

Figure 5 shows this empirically, as it takes a very similar time to launch configurations with the same total number of nodes, but extremely different mixed of Config Servers, Mongos Instances, Replicas, and Shards.

The total number of nodes in a deployment can be calculated by the following equation involving the parameters to the deployment script.

c + m + (s * r) = total nodes

Config Servers	Mongos	Shards	Replicas	Time in
-c	-m	-s	-r	Seconds
5	1	1	1	1258
1	5	1	1	1253
1	1	5	1	1292
1	1	1	5	1277

Fig. 5. Deployment Times on Chameleon Cloud in Seconds

Note: Due to the floating IP issues in the Kilo environment, all deployment times shown are for Chameleon environment. Professor von Laszewski is aware of these infrastructure issues and they cannot be resolved by the project team. Fugang Wang is currently working on resolving them.

9. BENCHMARKING

After the sharded MongoDB instance has been fully deployed, a benchmarking process is run to assess performance of the configuration. This process has also been fully automated. It is automatically invoked after the deployment when running mongo_deploy.sh. An inventory of the code used to automate the benchmarking process is provided in Appendix B.

9.1. Data Set

Add Map/Reduce

The data set used in the benchmarking testing and analysis was Major League Baseball PITCHf/x data obtained by using the program Baseball on a Stick (BBOS). [21] BBOS is a python program created by *willkoky* on github which extracts data from mlb.com and loads it into a MySQL database. While it would be possible to convert this program to populate the MongoDB database directly, collecting all of the data is a time consuming process. Therefore, the data was captured locally to the default MySQL database and then extracted to a CSV file. This file contains 5,508,014 rows and 61 columns. It is 1,588,996,075 bytes in size uncompressed.

9.2. Methodology

Will make major modications

The primary benchmarking goal of the project was to assess the impact of sharding on performance in MongoDB. Since replication was also built into the deployment process, a secondary goal was to assess the impact of replica sets on performance. The benchmarking tests are design to assess performance of the following situations.

MAY ADD MAP REDUCE TEST SCENARIO

https://docs.mongodb.com/manual/core/map-reduce/
https://docs.mongodb.com/manual/core/map-reduce-sharded-collections/

1. Impact of Sharding Configuration on Reads

2. Impact of Sharding Configuration on Writes

3. Impact of Replication Configuration on Reads

4. Impact of Replication Configuration on Writes

To access the impact of different configurations on writes, we use MongoDB's mongoimport command. Mongoimport is a command line tool capable of loading JSON, CSV, or TSV files. [22] In this case, we load a CSV file to the pitches collections in the mlb database.

To assess the impact of different configurations on reads, we use MongoDB's find command. We read the data previously loaded by the mongoimport command to the pitches collection. The find command retrieves documents that meet a specified criteria. In this case, we search for pitches with a speed over 100 mph, a relatively rare event in baseball. To limit the information sent back over the network, we only return a count of these events. 3,632 is the count returned of 5,508,014 total documents. The column we search on does not have an index, as the goal is to test the impact of sharding on a long running query.

For each test, timing results were captured in file benchmark_datetime.csv. This file included the configuration the MongoDB configuration the test was run under (cloud, config server replication factor, mongos instances, number of shards, and shard replication factor) along with the run times of the find and mongoimport commands. After all tests were run, a shell script, combine_benchmark.sh was run to combine all files into one file, benchmark_combined.csv.

The graphical depictions of the test results show in the next section were created by running python programs to average the run times across the shard and replication configurations shown. For consistency, config server replication and mongos instances were both kept at 1 for all benchmarking tests. Additionally, replication was kept at 1 for sharding tests and sharding at 1 for replication tests. This methodologies allows us to isolate the variable we are assessing.

A compressed version of file has been placed in an Amazon Web Services S3 directory. The benchmarking script retrieves the file via wget, uncompresses it, and loads it to a collection named *pitches* in MongoDB using mongoimport before running the find command.

9.3. Benchmarking Process

Before the benchmarking process can be run, a sharded collection must be created. The script mongo_collection.sh creates and shards a collection for the data set to be loaded to. As with other steps in the process, this is automated and runs out of mongo_deploy.sh.

Create Sharded Collection

1. Ssh to a mongos instance. Authenticating as user1 via the mongo shell, create the pitches collection.

2. Ssh to a mongos instance. Authenticating as cluster_admin_user via the mongo shell, run a command to enable sharding in the mlb database the pitches table is located in.

3. Ssh to a mongos instance. Authenticating as cluster_admin_user via the mongo shell, run a command to shard the collection.

The shard key is set to pitchID. PitchID is a unique key to each pitch document. Selecting pitchID as the shard key should cause the data to be reasonably evenly distributed amound the shards. Data distribution will be analyzed in a subsequent section.

Once the sharded collection has been created, benchmarking test can be run. The script mongo_benchmark.sh runs the benchmarking process. It adds the benchmarking times for both find and mongoimport to the file created during the deployment containing the configuration parameters and deployment time.

Benchmarking

1. Ssh to a mongos instance. Using wget retrieve the pitches file from a Amazon Web Services (AWS) Simple Storage Service (S3) folder and uncompress the file.

2. Connect to a mongo shell on the mongos instance and run a mongoimport of the file to the sharded collection, capturing the run time.

3. Connect to a mongo shell on the mongos instance and run a find command against the sharded collection, capturing the run time.

9.4. Data Distribution

To explore how data was allocated among the shards, the getShardDistribution() command, which reports on how data and documents are distributed among shards [23], is run as part of the automated process after the data is populated in the collection by mongoimport. Figure 6 show the results of tests with 1 to 5 shards. The results clearly show the data is well distributed, although interestingly, in all cases there is some skew toward the final shard having the most data.

	1	2	3	4	5
1	100				
2	45.83	54.16			
3	31.17	31.05	37.76		
4	23.72	23.98	24.14	28.14	
5	19.27	18.88	18.86	18.89	24.08

Fig. 6. Data Distribution among Shards

9.5. Benchmarking Analysis

9.5.1. Impact of Sharding on Reads

Will make major modications

Figure 7 depicts the impact on performance of various numbers of shards on a find command in both Chameleon and Kilo Clouds. The sharp decline in run time as the number of shards is increased shows the positive effect of sharding on performance. For example, while the run time for 1 shard in both Chameleon and Kilo is over 120 seconds, the time drops to around 15 seconds for 6 shards. This is a significant gain in performance.

For small numbers of shards, performance gains are almost exact proportion to the number of shards. 2 shards yields 1/3 the run time of 1 shard. 3 shards yields 1/2 the run time of 1 shard. From 3 to 6 chards, we still see still see significant improvement, but incrementally less than for the first 3 shards. After 6 shards, we see only slight performance gains.

From the closeness of the Chameleon and Kilo lines we can see that performance in the two clouds is very similar for this find test. This is an interesting observation as for both deployment and mongoimport, performance was much better on
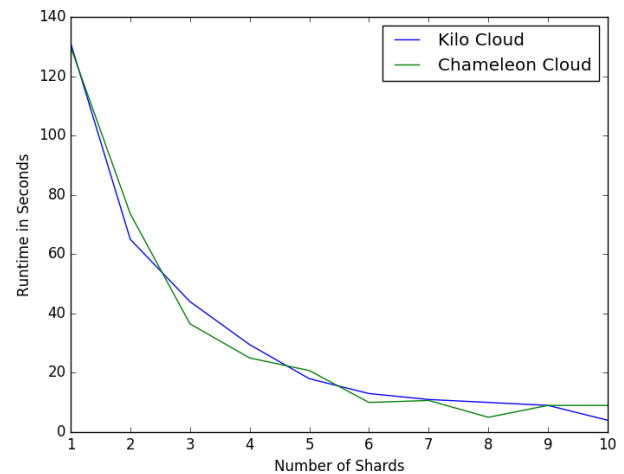

Fig. 7. Find Command - Sharding Test

Chameleon Cloud than Kilo. One difference from the mongoimport test is that much less data is being sent over the network. Network speeds could be a factor in this discrepency.

Figure 7 can be recreated by running the program benchmark_shards_find.py passing the file benchmark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib.

9.5.2. Impact of Sharding on Writes
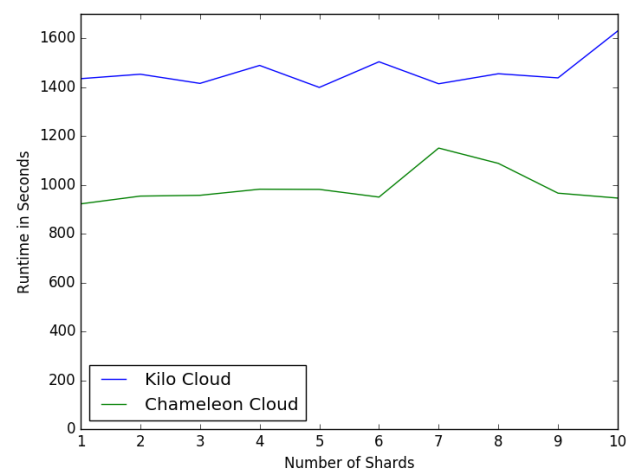
Will make major modications

Fig. 8. Mongoimport Command - Sharding Test

Figure 8 depicts the impact on performance of various numbers of shards on a mongoimport command in both Chameleon and Kilo Clouds. For both clouds run time of the mongoimport command in our tests does not appear to be impacted by the number of shards. Since the same amount of data is written with more computing resources available when there are more shards, we might expect to see a performance gain. However, there are possible explanations for performance not improving. First, the mongoimport command may not write data in parallel. This is not indicated in the documentation, but it seems likely that it

reads the file serially. Second, resources on the server the data is written to may not be the bottleneck in the write process. Other resources like the network time are likely to be the bottleneck. Since we are always going over the network from the mongos instance to a data shard, regardless of the number of shards, a bottleneck in the network would impact all shard configurations equally.

While sharding did not benefit a single threaded mongoimport command, it is likely it would benefit heavy write operations, particularly coming through multiple mongos instances. In a non-sharded environment, this would lead to a heavy load on the single data shard. In a sharded environment, the load on each shard would drop as the number of shards increased.

While performance on Chameleon and Kilo was very similar for the find command, performance of the mongoimport command was significantly better on Chameleon than on Kilo. We see approximately 50% better performance on Chameleon Cloud than on Kilo. Although not formally plotted due to environmental issue in Kilo Cloud, similar results were observed for deployment times. Overall, Chameleon Cloud is a faster environment that FutureSystems Kilo Cloud.

Figure 9 can be recreated by running the program benchmark_shards_import.py passing the file benchmark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib.

9.5.3. Impact of Replication on Reads
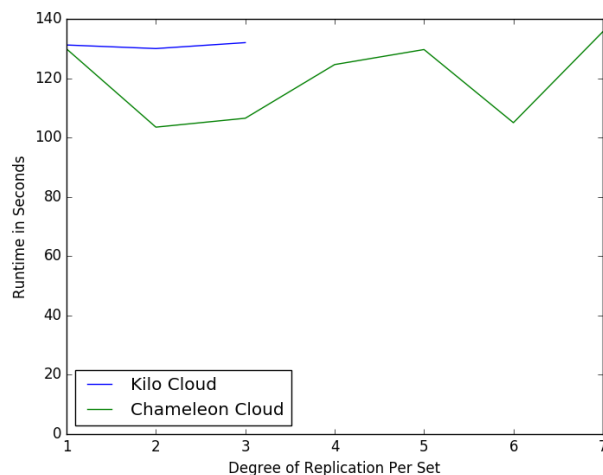
Will make major modications

Fig. 9. Find Command - Replication Test

Figure 9 depicts the impact on performance of various numbers of replicas on a find command in both Chameleon and Kilo Clouds. (Note: Due to environmental issues with floating IPs in the India/Kilo environment, tests were only able to be run from 1, 2, and 3 replica configurations on Kilo cloud. This is an infrastructure problem beyond the control of the project team that Professor von Laszewski is aware of.) While it is clear that replication does not have the same performance impact on the find command that sharding does, it appears that there may be a slight performance penalty to high degrees of replication. Replica sets up to 3 did not show this penalty, but 4 and 7 replica sets both had increased run times. Without a larger sample size it cannot be determined if this is a real effect or random variation.

We would not expect replication to have a significant performance degradation on the the find command since it only needs to read one copy of the data, but the increased communication necessary in a replica set may cause a small performance penalty in some cases.

Similarly to our sharding mongoimport test, performance on Chameleon was better than on Kilo for all test runs in the find replication test. The difference was similar to the 50% improvement noted earlier.

Figure 9 can be recreated by running the program benchmark_replicas_find.py passing the file benchmark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib.

9.5.4. Impact of Replication on Writes
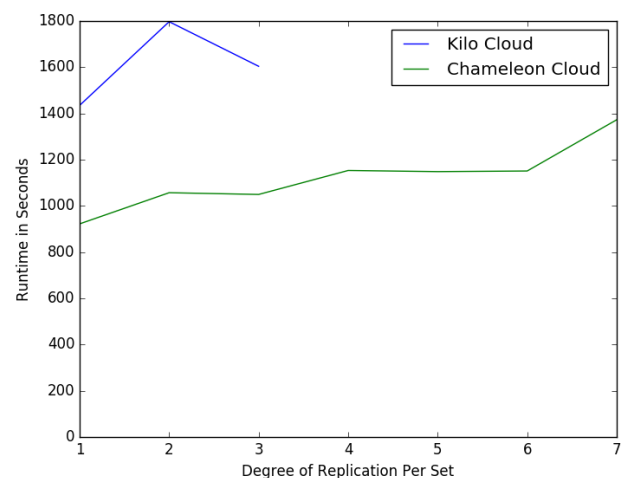
Will make major modications

Fig. 10. Mongoimport Command - Replication Test

Figure 10 depicts the impact on performance of various numbers of replicas on a mongoimport command in both Chameleon and Kilo Clouds. (Note: Due to environmental issues with floating IPs in the India/Kilo environment, tests were only able to be run from 1, 2, and 3 replica configurations on Kilo cloud. This is an infrastructure problem beyond the control of the project team that Professor von Laszewski is aware of.) The test results show negative impact on the mongoimport command for increase levels of replication. On Chameleon, a replication factor of 2 leads to approximately a 20% performance penalty and a replication factor of 7 leads to 50% worse performance. Given that an extra copy of data is written with each increase in the replication factor, this performance hit is expected.

Similarly to our sharding mongoimport and find replication tests, performance on Chameleon was better than on Kilo for all test runs in the mongoimport replication test, again by a similar percentage.

Figure 10 can be recreated by running the program benchmark_shards_import.py passing the file benchmark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib.

10. SUMMARY

Will make major modications

We have created, tested, and demonstrated a fully automated program to configure and deploy a sharded MongoDB cluster to three cloud environments. The cluster can be deployed with a selected number of Config Server Replicas, Mongos Routers, Shards, and Shard Replicas. An automated benchmarking process to show the impact of well distributed data across shards of a large data set has been created and run for various configurations to assess the impact of Sharding and Replication on performance. A key finding of these tests is that read performance, typically a high priority for noSQL data stores, increases significantly as shards are added. Testing also showed a predictable performance penalty is associated with replication.

ACKNOWLEDGEMENTS

REFERENCES

[1] Wikipedia, "Openstack," web page, Nov. 2016, 11/9/2016. [Online]. Available: https://en.wikipedia.org/wiki/OpenStack

[2] OpenStack, "Openstack community q&a," web page, Nov. 2016. [Online]. Available: https://www.openstack.org/projects/openstack-faq/

[3] Chameleon, "About," web page, Nov. 2016. [Online]. Available: https://www.chameleoncloud.org/about/chameleon/

[4] FutureSystems, "About," web page, Sep. 2014. [Online]. Available: https://portal.futuresystems.org/about

[5] ——, "Staff," web page, Dec. 2016. [Online]. Available: https://portal.futuresystems.org/staff

[6] Chameleon, "Hardware description," web page, Nov. 2016. [Online]. Available: https://www.chameleoncloud.org/about/hardware-description/

[7] G. von Laszewski, "Hardware," web page, Jan. 2013. [Online]. Available: http://futuregrid.github.io/manual/hardware.html

[8] ——, "Cloudmesh client toolkit," web page, Sep. 2016. [Online]. Available: http://cloudmesh.github.io/client/

[9] Wikipedia, "Mongo," web page, Sep. 2016. [Online]. Available: https://en.wikipedia.org/wiki/MongoDB

[10] DB-Engines, "Bdb-engines ranking," web page, Nov. 2016. [Online]. Available: http://db-engines.com/en/ranking

[11] MongoDB, "Sharded cluster components," web page, Sep. 2016. [Online]. Available: https://docs.mongodb.com/v3.2/core/sharded-cluster-components/

[12] ——, "Config servers," web page, Nov. 2016. [Online]. Available: https://docs.mongodb.com/manual/core/sharded-cluster-config-servers/

[13] ——, "Mongos," web page, Nov. 2016. [Online]. Available: https://docs.mongodb.com/manual/reference/program/mongos/

[14] ——, "Sharding," web page, Sep. 2016. [Online]. Available: https://docs.mongodb.com/manual/sharding/

[15] ——, "Shard keys," web page, Sep. 2016. [Online]. Available: https://docs.mongodb.com/manual/core/sharding-shard-key/

[16] ——, "Replication," web page, Sep. 2016. [Online]. Available: https://docs.mongodb.com/manual/replication/

[17] ——, "Replica set deployment architectures," web page, Nov. 2016. [Online]. Available: https://docs.mongodb.com/v3.2/core/replica-set-architectures/

[18] ——, "Enable internal authentication," web page, Nov. 2016. [Online]. Available: https://docs.mongodb.com/v3.0/tutorial/enable-internal-authentication/

[19] ——, "Deploy a sharded cluster," web page, Sep. 2016. [Online]. Available: https://docs.mongodb.com/v3.2/tutorial/deploy-shard-cluster/

[20] M. Sanaulla, "Setting up sharded mongodb cluster in localhost," web page, Feb. 2015.

[21] willkoky, "Baseball on a stick," web page, Apr. 2016. [Online]. Available: https://sourceforge.net/projects/baseballonastic/

[22] MongoDB, "mongoimport," web page, Sep. 2016. [Online]. Available: https://docs.mongodb.com/manual/reference/program/mongoimport/

[23] ——, "db.collection.getsharddistribution()," web page, Nov. 2016. [Online]. Available: https://docs.mongodb.com/manual/reference/method/db.collection.getShardDistribution/

AUTHOR BIOGRAPHIES
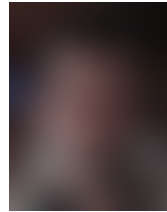
Mark McCombe received his B.S. (Business Administration/Finance) and M.S. (Computer Information Systems) from Boston University. He is currently studying Data Science at Indiana University Bloomington.

A. WORK BREAKDOWN

The work on this project was distributed as follows between the authors:

Mark McCombe. Completed all work.