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 author: mmccombe@iu.edu*

Using Python, Ansible, Bash Shell, and Cloudmesh Client a fully automated process is created for deploying a configurable MongoDB sharded cluster on Chameleon, FutureSystems, and Jetstream cloud computing environments. A user runs a single Python program which configures and deploys the environment based on parameters specified for numbers of Config Server Replicas, Mongos Instances, Shards, and Shard Replication. The process installs either MongoDB version 3.4 or 3.2 as requested by the user. Additionally, functionality exists to run benchmarking tests for each deployment, capturing statistics in a file as input for python visualization programs, the results of which are displayed in this report. These reports depict the impact of MongoDB version and degrees of sharding and replication on performance. Key performance findings regarding version, sharding, and replication are abstracted from this analysis. 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

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

1. INTRODUCTION

As the final project for I524, Big Data Software and Projects, Spring 2017, a Python program invoking Bash shell scripts, Ansible playbooks, and Cloudmesh Client commands 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 cloud environments. The scripts have been developed and tested on an Ubuntu 16.04 LTS (Xenial Xerus) Virtual Machine running in Virtual Box. Using the Python cmd command line interface, the program project.py accepts parameters for deployment cloud, MongoDB version, Config Server server replication, number of Mongos instances, number of Data Shards, and Shard replication.

Also via project.py, automated benchmarking tests can be run. Tests were performed with various sharding and replication configurations to assess their impact on performance. Additionally, tests were run against MongoDB versions 3.4 and 3.2 to uncover any performance differences between these version. Performance results are captured and graphed using Python's matplotlib, the results of which are displayed and analyzed in this report.

2. INFRASTRUCTURE

Three clouds were selected for deployment: Chameleon Cloud, Futuresystems (also referred to as Kilo in some sections of this document), and Jetstream. In our automated deployment and benchmarking process, the cloud name is passed as a parameter to the deploy function of the main project.py script and a customized version of MongoDB is deployed to the selected cloud.

2.1. OpenStack

Chameleon Cloud, FutureSystems and Jetstream all utilize 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].

2.2. 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].

2.3. 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 [4]. For our deployment, we utilize the OpenStack Kilo Cloud, running on the India machine. Because the environment is by default referred to as Kilo in the Cloudmesh documentation and setup file, it is referred to as both FutureSystems and Kilo in subsequent sections of this document and the accompanying diagrams.

2.4. Jetstream

Jetstream is a cloud computing environment implemented by many academic and industry partners including the University of Texas at Austin's Texas Advanced Computing Center (TACC), the Computation Institute at the University of Chicago, the University of Arizona, the University of Texas San Antonio, Johns Hopkins University, Penn State University, Cornell University, the University of Arkansas at Pine Bluff, the National Snow and Ice Data Center (NSIDC), the Odum Institute at the University of North Carolina, the University of Hawaii, and Dell [5]. At Indiana University, leadership is provided by the Pervasive Technology Institute with involvement from several members of the School of Informatics and Computing including Beth Plale, Katy Borner, and Volker Brendel [6].

2.5. Cloud Hardware Comparison

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

Table 1. Cloud Hardware Specification Comparison [7] [8] [9]

	FutureSystems	Chameleon	Jetstream
CPU	Xeon E5-2670	Xeon X5550	Haswell E-2680
cores	1024	1008	7680
speed	2.66GHz	2.3GHz	2.5GHz
RAM	3072GB	5376GB	40TBr
storage	335TB	1.5PB	2 TB

3. PYTHON/CMD

Python is utilized in two portions of the automated process. First, the main script, project.py, is a Python program that utilizes the cmd module to provide a simple command line interface [10]

accepting parameters for deployment configuration. project.py also provides other functionality such as cluster deletion, benchmarking, benchmarking summarization and reporting, and data distribution reporting. Second, several visualization programs for benchmarking analysis are written in Python, utilizing the matplotlib and pandas modules.

4. ANSIBLE

Ansible is open source software typically use to automate software provisioning and configuration management. Ansible uses Playbooks specified in YAML file format to accomplish this goal. Ansible runs on Linux/Unix and requires Python [11].

In our deployment, virtual machines are created using Cloudmesh Client cluster commands. Once they are created, all direct cloud interaction for the MongoDB software installation and environment customization and setup is performance via Ansible playbooks.

5. CLOUDMESH CLIENT

The Cloudmesh Client toolkit is an open source client interface that standardizes access to various clouds, clusters, and workstations [12]. Cloudmesh Client is a python based application developed by Gregor von Laszewski and others collaborators primarily at Indiana University.

In the deployment, Cloudmesh Client is used to handle most interaction with the Virtual Machines in the clouds. Cloudmesh Client provides functionality in three main areas: Key Management, OpenStack Security, and virtual machine management. For key management, Cloudmesh's key add and upload commands simplify secure interaction with the cloud environments. For Openstack security, Cloudmesh's secgroup commands allow new security rules to be added and uploaded to the cloud. Virtual machine management is performed with Cloudmesh's cluster functionality, which allows easy creation and deletion of virtual machines and communication between them.

Cloudmesh Client simplifies and standardized interaction with the cloud for these tasks. This allows us to more easily port the deployment to additional clouds that are supported by Cloudmesh. Furthermore, by encapsulating the logic necessary to perform these tasks we are shielded from changes in interfaces made by individual clouds.
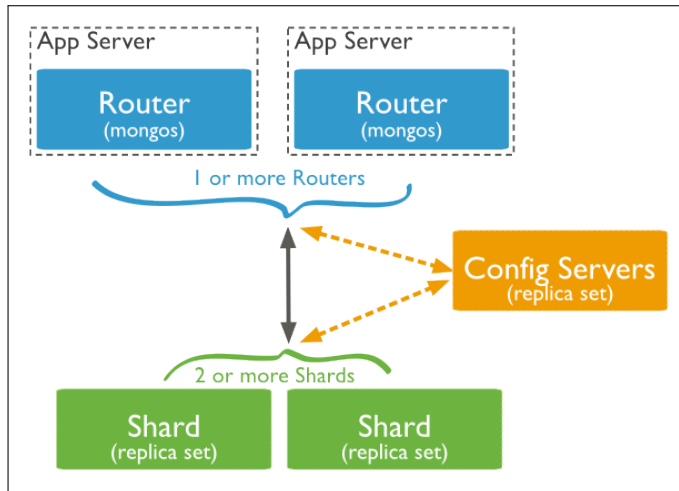
6. MONGODB

MongoDB is a popular open source, document oriented noSQL database. It stores documents in JSON-like schema-less formats called collections [13]. DBEngines ranks MongoDB as the most popular noSQL store and as the fifth most popular Database Management System overall [14].

6.1. 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

Figure 1 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.

Fig. 1. Sharded MongoDB Architecture [15]

6.2. 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 [16].

Config Servers also contain authentication information. For example, information about the keyfiles used for internal authentication between the nodes is stored in the Config Servers [16].

In production deployments, it is recommended for Config Servers to be deployed in 3 member replica sets [15]. 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 by the third parameter to the main project.py script. For example, specifying 1 will create a Replica Set with one Config Servers (not replication), specifying 3 will create a Replica Set with three Config Server, and so on.

6.3. 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 [17].

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 [16].

In our deployment and benchmarking automation, the number of Mongos instances is controlled by the fourth parameter to the main project.py script. For example, specifying 1 will create one Mongos instance, specifying 3 will create three Mongos instances, and so on.

6.4. Shards

Sharding, or distributing data across machines, is used by MongoDB to support large data sets and provide high throughput [18]. 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 [18]. Since the shard key is critical to performance and efficiency, particular care must be given to shard key selection [19]. 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 fifth parameter to the main project.py script. For example, specifying 1 cause only 1 shard to be created, specifying 3 will cause three shards to be created, and so on.

6.5. Replication

In databases, replication provides data redundancy leading to greater availability and fault tolerance. Replication in MongoDB is achieved via Replica Sets [20]. Replica Sets were implemented in our deployment for both Config Servers and Shards.

Two key benefits provided by replication are redundancy and fault tolerance. Each Replica in a Replica Set provides another copy of data. Higher redundancy means more nodes can be lost without data being lost. Higher numbers of Replicas in a set also 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

Table 2. Fault Tolerance by Replica Set Size [21]

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

As shown in Table 2, 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 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 [21].

For production systems, a standard deployment is a 3 Replica Set [21]. 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 sixth parameter to the main project.py script. For example, specifying

1 for will create a replica set per shard with only one copy of data (essentially no replication, although technically we create a one member replica set), specifying 3 will cause a replica set of with three copies to be created, and so on.

6.6. MongoDB Versions

The most current version of MongoDB is version 3.4, which was released on November 29, 2016. Based on an input parameter, our deployment will install either version 3.4 or version 3.2, the prior version of MongoDB. Many enhancements were made for version 3.4 impacting Sharded Clusters, Replica Sets, Aggregation, Indexes, Views, Security, Tools, and Platform Support. The complete list of 3.4 features and enhancements can be found in the Release Notes [22].

In our automated deployment and benchmarking process, the version of MongoDB installed is controlled by the second parameter to the main project.py script. Specifying 34 will install version 3.4. Specifying 32 will install 3.2. These versions were selected as they are the two most recent major versions of MongoDB and because they are the only two compatible with Ubuntu 16.04 LTS (Xenial Xerus).

6.7. 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 [23]. In our deployment, authentication is done by keyfiles. A new keyfile is automatically created for each deployment and distributed to all of the virtual machines on the selected cloud.

For external authentication, the three users are created. The user *admin* is created with the role of *userAdminAnyDatabase*. *admin* performs administrative functions such as creating the other users. The user *cluster_admin_user* is created with the role *clusterAdmin*. *cluster_admin_user* user performs sharding functions such as sharding the collection and checking its data distribution. *user1* is a standard user with readWrite permissions. *user1* performs the benchmarking tests and other functions not requiring administrative privileges.

7. DEPLOYMENT

The automated process fully deploys a sharded MongoDB environment with the Cloud, MongoDB version, number of Config Servers, Mongos Instances, Shards, and degree of Replication specified as input parameters.

7.1. Computing Resources

In all cases, virtual machines are deployed with the Ubuntu 16.04 LTS (Xenial Xerus) operating system. On Openstack the flavor or the machine determines the amount of computing resources (CPU, memory, storage) allocated to it. In our testing, m1.medium was used as the flavor for Chameleon Cloud and FutureSystems, while m1.small was used on Jetstream. Jetstream has more resources allocated to each flavor than Chameleon and

FutureSystems, which are similar. In order to perform similar tests on each cloud, flavors with identical CPU and memory were selected. Table 3 shows the comparative resources of the flavors used in our testing. While storage is lower on Jetstream, it is sufficient for out tests and should not significantly impact performance.

Table 3. Computing Resources

Cloud	Flavor	VCPU	RAM	Size
Chameleon	m1.medium	2	4	40
FutureSystems	m1.medium	2	4	40
Jetstream	m1.small	2	4	20

7.2. Deployment Process

Several programs are involved in the deployment. A high level overview of each is provided.

7.2.1. project.py

The deployment process is invoked by running the deploy function of project.py, passing six required parameters: cloud (chameleon, jetstream, or kilo), version (32 or 34 for version 3.2 or 3.4), size of the config server replica set (a number, 1 or greater), number of mongos routers (a number, 1 or greater), number of data shards (a number, 1 or greater), and size of data shard replica sets.

Project.py calls a bash script, deploy.sh, which runs two bash shell scripts to accomplish the deployment: cluster.sh and ansible.sh.

7.2.2. cluster.sh

Cluster.sh does the work of creating the cluster in the specified cloud environment. First, it creates a keyfile needed for secure access between the nodes, and uses Cloudmesh secgroup commands to builds and uploads a new security group with the ports necessary for MongoDB (27017, 27018, 27019, and 28017) accessible. Next, it uses Cloudmesh client cluster commands to launch the appropriate number of virtual machines in the desired cloud. Then, it builds a file, inventory.txt, with sections for each MongoDB componenent (Config Servers, Mongos Instances, and Shard Replica Sets), allocating the correct number of IP addresses to each. Finally, cluster.sh builds a few complex commands that will need to be run later in the process by ansible.

7.2.3. ansible.sh

After the virtual machines have been created by cluster.sh, ansible.sh used the inventory.txt file to execute Ansible playbooks on the appropriate virtual machines.

1. install-python.yaml - Installs Python, if not installed. This script was necessary because the Ubuntu Zenial image on FutureSystems does not have Python installed. Python is required for Ansible.

2. mongo-install32.yaml - Using apt_key and apt_repository, installs the packages for version 3.2 of MongoDB on all virtual machines.

3. mongo-install34.yaml - Using apt_key and apt_repository, installs the packages for version 3.4 of MongoDB on all virtual machines.

4. add-mongo-key.yaml - Uploads the key file created in cluster.sh to all virtual machines.

5. mongo-config.yaml - On Config Servers only, stops the mongod service and uses a template file to start the mongod process for a Config Server.

6. mongo-config2.yaml - On only one Config Server, uses a template file to initiate the primary Config Server.

7. mongo-mongos.yaml - On Mongos Instances only, stops the mongod service and uses a template file to start the mongos process for a Mongos instance.

8. mongo-users.yaml - On only one Mongos Instance, create several users needed in later steps.

9. mongo-shard.yaml - On Shards only, stops the mongod service and uses a template file to start the mongod process for a Shard.

10. mongo-shard2.yaml - On the primary Shard in each Replica Set, uses a file built in cluster.sh to initiate the Shards.

11. add-shards.yaml - On one Mongos instance, uses a file built in cluster.sh to add all of the Shards.

12. create-sharded-collection.yaml - Uploads several files to one Mongos instance that will be need for benchmarking and shard the collection (benchmarking setup, not included in deployment times).

13. getdata.yaml - Downloads and unarchivse the pitches data from an AWS S3 directory. Also, create a smaller version for testing (benchmarking setup, not included in deployment times).

The kill function in project.py will delete and deallocate the last existing cluster on the cloud to clean up after the test is complete.

7.3. Deployment Timing

The configuration parameters and cluster and Ansible deployment times are captured in a file for each deployment (benchmarking timings are later captured as well). Total run time for a few interesting configurations are shown in Table 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 completed in 330 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 took 1059 seconds to deploy.

Deployment C shows a deployment focused on high performance. It has a high number of shards, nine, but no fault tolerance or redundancy. The deployment may be suitable where performance needs are high and availability is less critical. Deployment C finished in 719 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. The additional steps to configure Config Servers, Mongos Instances, Replicas, and Shards run in relatively similar times, so

Table 4. Deployment Times on Chameleon Cloud in Seconds

	Config	Mongos	Shards	Replicas	Seconds
A	1	1	1	1	*330*
B	3	2	3	3	*1059*
C	1	1	9	1	*719*

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 five Config Servers took 534 seconds, one with five Mongos Instances took 556 seconds, one with five Shards took 607 seconds, and one with a five Shard Replica set took 524 seconds. There is small extra overhead to starting additional data shards, but a strong correlation exists for total nodes to runtime for all configurations. Deployment times for version 3.4 were very similar to version 3.2.

Table 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

Table 5. Deployment Times on Chameleon Cloud in Seconds

Config Servers	Mongos	Shards	Replicas	Time in
-c	-m	-s	-r	Seconds
5	1	1	1	*534*
1	*5*	1	1	*556*
1	1	*5*	1	*607*
1	1	1	*5*	*524*

Due to Chameleon Cloud having the most reliable and consistent performance of the three clouds, performance numbers are presented only for selected runs on Chameleon. These numbers are proportionately representative of deployment timings on Jetstream and FutureSystems.

8. 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 invoked by running the benchmark function of project.py and passing either the parameter large (for a full benchmark test) or small for a small test.

8.1. Data Set

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) [24]. 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.

8.2. Methodology

There are several goals of the benchmarking process. The primary benchmarking goal of the project is 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. A third goal is to assess performance of MongoDB version 3.4 versus version 3.2, specifically for various shard configuration. A final objective is to assess the relative performance of the Chameleon, FutureSystems, and Jetstream cloud computing environments.

The benchmarking tests are design to assess performance in three situations: Reads, Writes, and MapReduce operations.

8.2.1. Impact on Reads

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 [25]. In this case, we load a CSV file to the pitches collections in the mlb database.

8.2.2. Impact on Writes

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.

8.2.3. Impact on MapReduce

To assess the performance of MongoDB version, sharding, and replication on reads, a simple MapReduce operation was written against the pitches table to get the average speed of pitches that were stikes versus those that were not strikes [26] [27].

8.3. Benchmarking Process

The benchmarking process is invoked by running the benchmark function of the script project.py with the large parameter. Results for each test are automatically captured in file benchmark_datetime.csv. This file included the configuration the test was run under (cloud, MongoDB version, config server replication factor, mongos instances, number of shards, and shard replication factor) along with the run times of the find, mongoimport, and MapReduce commands. After all tests were run, a shell script, combine_benchdeploy.sh combines all files into one file, benchmark_combined.csv.

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

All sharding tests were run with one, two, three, five, seven, and nine shards. All replication tests were run with one, two, three, five, and seven replicas.

To setup for the test, a compressed version of file has been stored in an Amazon Web Services S3 directory. This file is prestaged on a Mongos instance during the deployment (but excluded from the run time) and is loaded it to a collection named *pitches* in MongoDB using mongoimport before running the find and MapReduce commands.

Before the benchmarking process can be run, a sharded collection must be created and sharded. This was also done via Ansible during the deployment in preparation for benchmarking. For benchmarking rerunability, the benchmarking process also deletes any data from the pitches collection that may have been loaded prior to running Mongoimport.

The shard key for the pitches table 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.

8.4. Data Distribution

To explore how data was allocated among the shards, a function called distribution was built into project.py. This function runs the getShardDistribution() command, which reports on how data and documents are distributed among shards [28]. Tables 6 and 7 show the results of tests with one, three, and five shards in version 3.2 and 3.4 of MongoDB. The results clearly show the data is well distributed, although interestingly, in all cases there is some minor skew toward the first shard having the most data. These results clearly show that data distribution is similar in both versions of MongoDB.

Table 6. Data Distribution among Shards - Version 3.2

	1	2	3	4	5
1	100				
3	35.84	32.18	31.96		
5	23.04	19.27	19.40	19.38	18.89

Table 7. Data Distribution among Shards - Version 3.4

|	1	2	3	4	5
1	100				
3	36.23	31.82	35.75		
5	22.26	19.67	19.42	19.37	19.26

8.5. Benchmarking Analysis

8.5.1. Cloud Analysis

Chameleon Cloud was significantly more stable and reliable than FutureSystems and Jetstream Clouds for our testing. Chameleon yields the fastest and most consistent results with very few errors. Jetstream initially had stability problems that were eventually resolved by the Jetstream support team. Once these issues were resolved, Jetstream performance and stability

was very close to Chameleon's. FutureSystem performance was the poorest with respect to run time. Environmental errors were initially frequent, but after allocating new floating IPs test would be completed successfully. JetStream performance was good, but the environment was very unstable. Due to its stability and performance, Chameleon was chosen as the environment to test MongoDB version 3.4 versus 3.2, due to its stability.

8.5.2. Impact of Sharding on Reads

gure 2 depicts the impact on performance of various numbers of shards on a find command in Chameleon, FutureSystems, and Jetstream Clouds. All three clouds show a strong overall decline in run time as the number of shards increases, which shows the positive impact of sharding on performance. For all clouds, reads were over 35 seconds for one shard and less than 10 seconds for five shards. This is a significant gain in performance.
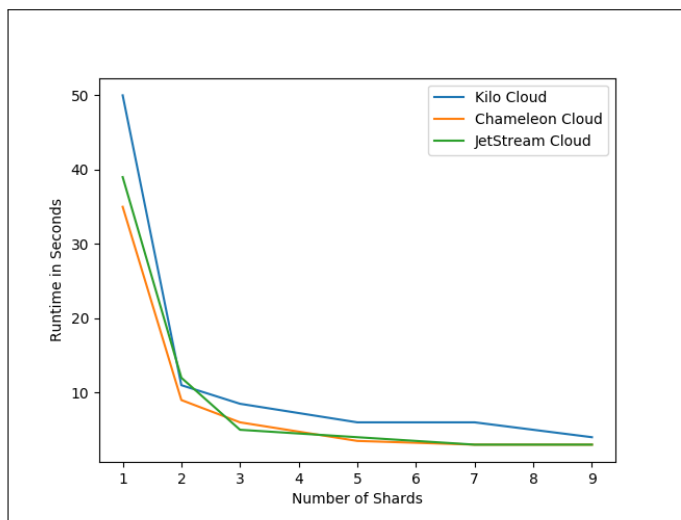
Fig. 2. Find Command - Sharding Test

All three clouds show a particularly large gain in performance when in increasing from one shard to two. Run time for two shards is less than one third the run time of one shard. Increases in shards beyond two show much smaller incremental gains.

Perfomance on Chameleon Cloud and Jetstream is very similar for the find test. Kilo performance is worse, although proportunately better than on the mongoimport test. This is an interesting observation as for both deployment and mongoimport, performance was much better on Chameleon and Futuresystems 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 discrepancy.

Figure 2 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. This report is run automatically by the report function of project.py.

8.5.3. Impact of Sharding on Writes

Figure 3 depicts the impact on performance of various numbers of shards on a mongoimport command in the three clouds. For all 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 seem more 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.
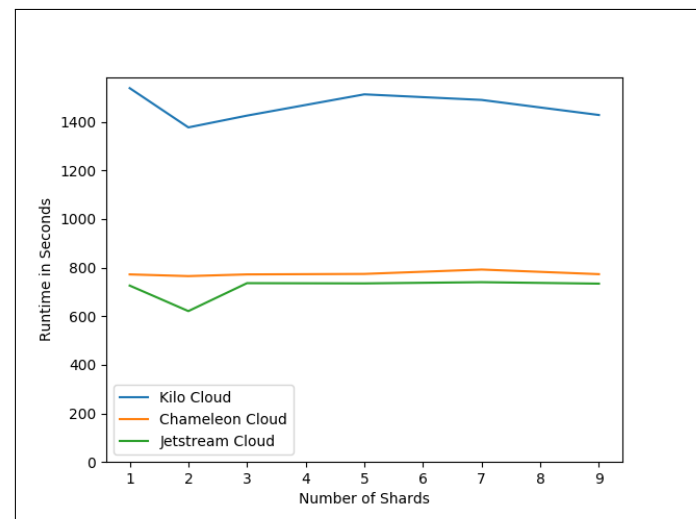
Fig. 3. Mongoimport Command - Sharding Test

While sharding did not benefit a single threaded mongoimport command, it is likely it would benefit other 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 FutureSystems 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 both Chameleon and Jetstream Clouds compared to FutureSystems. Jetstream performance is slightly better than Chameleon for the import test.

Figure 3 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. This report is run automatically by the report function of project.py.

8.5.4. Impact of Sharding on MapReduce

Figure 4 shows the performance of MapReduce across varous sharding configurations on our three clouds. These results are relatively similar to the find results. While results are inconsistent, particularly on Futuresystems, likely due to environmental issues, all clouds show an overall decrease in processing time with addition of shards. Relative to Mongoimport performance, performance is more similar across the three clouds for MapReduce.

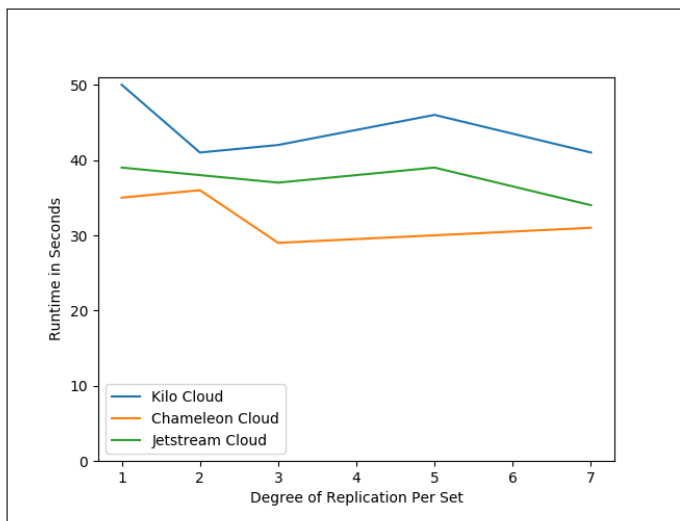Figure 4 can be recreated by running the program benchmark_shards_mapreduce.py passing the file bench-
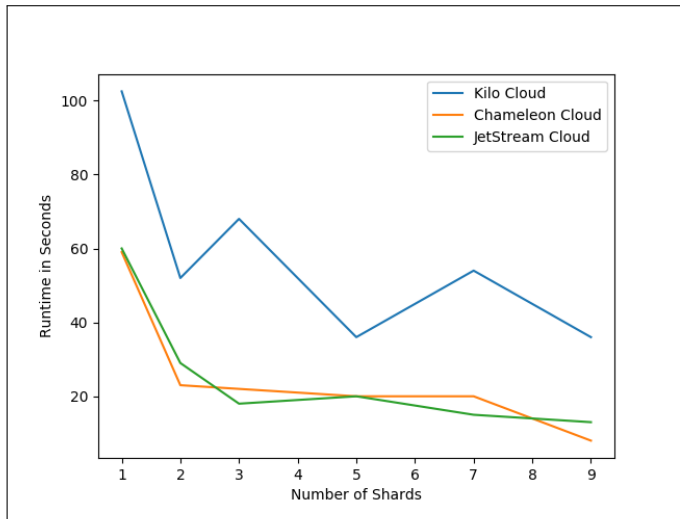
Fig. 4. MapReduce - Sharding Test

mark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib. This report is run automatically by the report function of project.py.

8.5.5. Impact of Replication on Reads

Figure 5 depicts the impact on performance of various numbers of replicas on a find command in Chameleon, FutureSystems, and Jetstream Clouds. These results show no correlation between the number of replicas and find performance.
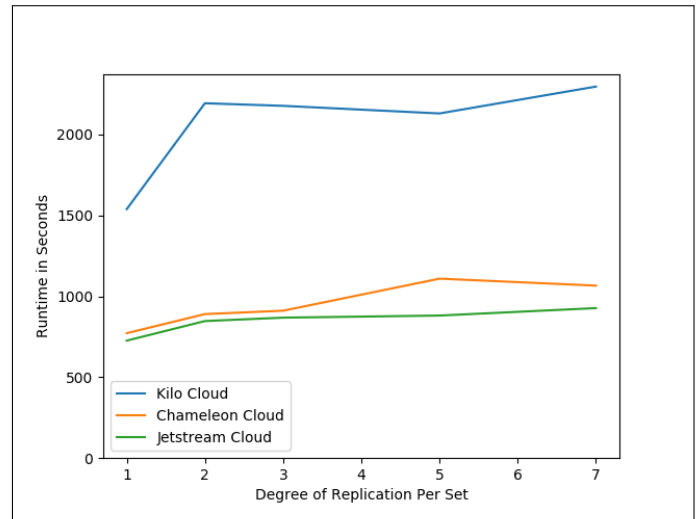

Fig. 5. Find Command - Replication Test

Similarly to other tests, performance on Chameleon was best for the majority of the test runs in the find replication test, followed by Jetstream, with Futuresystems performing the worst.

Figure 5 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. This report is run automatically by the report function of project.py.

8.5.6. Impact of Replication on Writes

Figure 6 depicts the impact on performance of various numbers of replicas on a mongoimport command on our three Clouds.


Fig. 6. Mongoimport Command - Replication Test

The results show poorer write performance as the number of replicas increase. Given that an extra copy of data is written with each increase in the replication factor, this performance hit is expected.

Performance on Jetstream and Chameleon were very close on this test with Chameleon only performing significantly better with four or more replicas. FutureSystems import performance was by far the worst of the three clouds.

Figure 6 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. This report is run automatically by the report function of project.py.

8.5.7. Impact of Replication on MapReduce
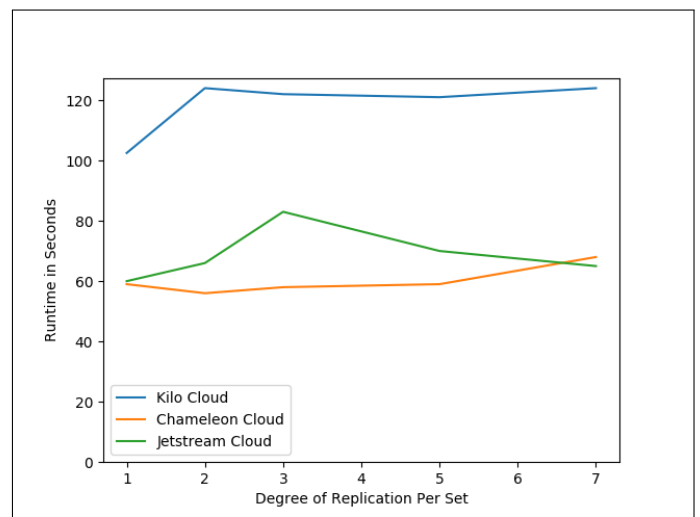
Fig. 7. MapReduce - Replication Test

As shown in Figure 7, replication appears to have no impact on MapReduce operations. While there are variations in FutureSystems and Jetstream performance for different numbers of replicas, they do not follow a consistent pattern and appear to be caused by environmental issues. This is an interesting result as

increased levels of replication came with a performance penalty for the find commmand, which also reads data.

As with several other tests, Chameleon MapReduce performance was the best, followed by Jetstream, with FutureSystems again being the worst.

Figure 7 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. This report is run automatically by the report function of project.py.

8.5.8. Impact of Version and Sharding on Reads
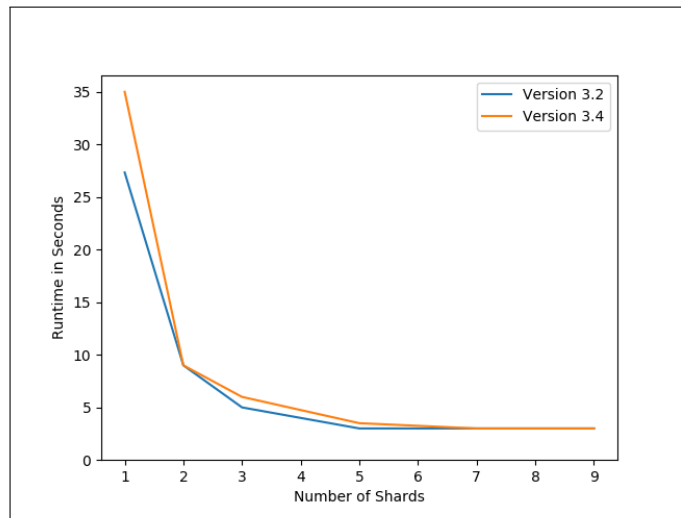
Fig. 8. Find Command - Version 3.2 vs 3.4

Figure 8 shows the MongoDB version 3.4 and 3.2 find performance on Chameleon Cloud. Results are very close, with version 3.2 having the best performance for one shard and performance being similar for all other sharding levels.

Figure 8 can be recreated by running the program benchmark_verson_find.py passing the file benchmark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib. This report is run automatically by the report function of project.py.

8.5.9. Impact of Version and Sharding on Writes

Figure 9 shows the MongoDB version 3.4 and 3.2 Mongoimport performance on Chameleon Cloud. Runtimes are similar for each version. Version 3.2 is slightly faster at the lowest sharding levels and Version 3.4 is slightly faster at the highest sharding level. Given the mixed results and close run times, neither version shows a significant advantage for write operations.
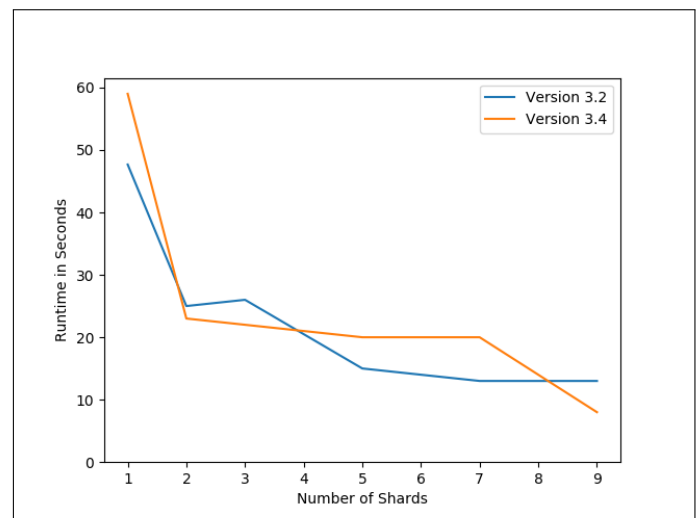
Figure 9 can be recreated by running the program benchmark_verson_find.py passing the file benchmark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib. This report is run automatically by the report function of project.py.

8.5.10. Impact of Version and Sharding on MapReduce

Figure 10 shows the MongoDB version 3.4 and 3.2 Mongoimport performance on Chameleon Cloud. Runtimes are similar for each version and with each version being faster at some shard level, which appears to be random. Given the mixed results and close run times, neither version shows a significant advantage for MapReduce operations.
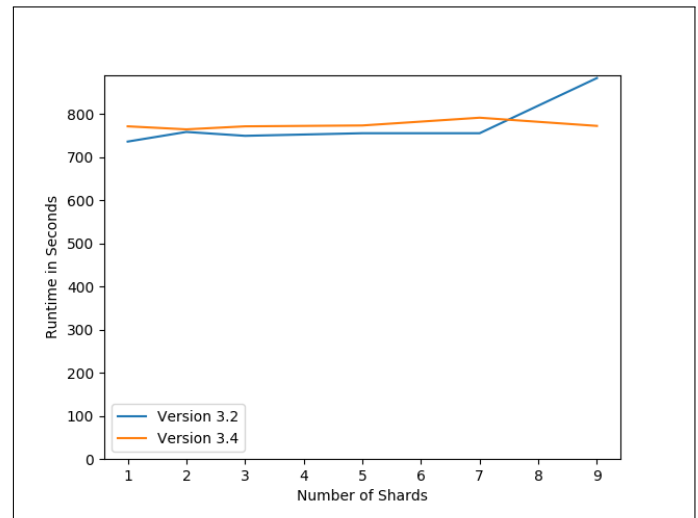
Fig. 9. Mongoimport Command - Version 3.2 vs 3.4

Fig. 10. MapReduce - Version 3.2 vs 3.4

Figure 10 can be recreated by running the program benchmark_verson_find.py passing the file benchmark_combined.csv as a parameter. It plots the average run time for each configuration as shown using matplotlib. This report is run automatically by the report function of project.py.

9. SUMMARY

We have created, tested, and demonstrated a fully automated program to configure and deploy a sharded MongoDB cluster to three cloud environments: Chameleon, Jetstream, and FutureSystems. Using a combination of Python, Bash, and Cloudmesh Client, the a cluster is dynamically deployed with a selected number of Config Server Replicas, Mongos Routers, Shards, and Shard Replicas and either MongoDB version 3.4 or 3.2. Functions also exist for terminating the environment, reporting on data distribution, benchmarking, and reporting on performance testing.

An automated benchmarking process to show the impact of well distributed data across shards of a large data set has been run for various configurations. The impact of MongoDB version 3.4 versus 3.2, Sharding, and Replication on performance have been assessed. Testing showed performance and stability on Chameleon Cloud to be the best of our three cloud environments with Jetstream a close second after an environmental issue was resolved by the support team. Futuresystems performance consistently lagged behind the other two clouds. A key finding is that read performance, typically a high priority for noSQL data stores and Big Data operations, increases significantly as shards are added. Testing also showed that a predictable performance penalty is associated with replication. Our comparison of version 3.4 and 3.2 showed no significant differences between version 3.2 and 3.4 performance across various sharding levels.

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] Indiana University, "Jetstream partners and collaborators," web page, 2016. [Online]. Available: http://www.jetstream-cloud.org/partners.php

[6] Indiana University, "Indiana university's leadership role," web page, 2016. [Online]. Available: http://www.jetstream-cloud.org/leadership.php

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

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

[9] JetStream, "System specs," web page, Apr. 2016. [Online]. Available: http://www.jetstream-cloud.org/leadership.php

[10] Python Software Foundation, "cmd — support for line-oriented command interpreters¶," web page, 4/20/2017. [Online]. Available: https://docs.python.org/2/library/cmd.html

[11] Wikipedia, "Ansible (software)," web page, Apr. 2017, 4/20/2017. [Online]. Available: https://en.wikipedia.org/wiki/Ansible_(software)

[12] G. von Laszewski, "Cloudmesh client toolkit," web page, Sep. 2016. [Online]. Available: http://cloudmesh.github.io/client/

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

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

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

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

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

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

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

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

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

[22] MongoDB, "Release notes," web page. [Online]. Available: https://docs.mongodb.com/manual/release-notes/3.4/

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

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

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

[26] MongoDB, "Mapreduce examples," web page. [Online]. Available: https://docs.mongodb.com/manual/tutorial/map-reduce-examples/

[27] MongoDB, "Mapreduce," web page. [Online]. Available: https://docs.mongodb.com/manual/core/map-reduce/

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

[29] C. Duffy, "How do i test if a variable is a number in bash?" web page, Apr. 2009. [Online]. Available: http://stackoverflow.com/questions/806906/how-do-i-test-if-a-variable-is-a-number-in-bash

[30] mklement0, "How can i remove the last character of a file in unix?" web page, Dec. 2014. [Online]. Available: http://stackoverflow.com/questions/27305177/how-can-i-remove-the-last-character-of-a-file-in-unix

[31] MongoDB, "Configuration file options," web page, Oct. 2016. [Online]. Available: http://docs.mongodb.org/manual/reference/configuration-options/

[32] steeldriver, "Passing named arguments to shell scripts," web page, May 2014. [Online]. Available: http://unix.stackexchange.com/questions/129391/passing-named-arguments-to-shell-scripts

[33] G. von Laszewski, "Advanced command usage," web page, Jan. 2015. [Online]. Available: http://cloudmesh.github.io/client/commands/command_advanced.html

[34] jezrael, "Converting a pandas groupby object to dataframe," web page, Aug. 2015. [Online]. Available: http://stackoverflow.com/questions/10373660/converting-a-pandas-groupby-object-to-dataframe

[35] silvio, "How to set 'auto' for upper limit, but keep a fixed lower limit with matplotlib.pyplot," web page, Sep. 2015. [Online]. Available: http://stackoverflow.com/questions/11744990

[36] EdChum, "Pandas dataframe to numpy array valueerror," web page, Aug. 2015. [Online]. Available: http://stackoverflow.com/questions/31791476/pandas-dataframe-to-numpy-array-valueerror

[37] leucos, "How to create a directory using ansible?" web page, Apr. 2014. [Online]. Available: http://stackoverflow.com/questions/22844905/how-to-create-a-directory-using-ansible

[38] El Ruso, "How to install mongodb with ansible?" web page, Feb. 2017. [Online]. Available: http://stackoverflow.com/questions/37568848/

how-to-install-mongodb-with-ansible

[39] Ansible, Inc., "copy - copies files to remote locations," web page, Apr. 2017. [Online]. Available: http://docs.ansible.com/ansible/copy_module.html

[40] Lorin Hochstein, "Run command on the ansible host," web page, Sep. 2013. [Online]. Available: http://stackoverflow.com/questions/18900236/run-command-on-the-ansible-host

[41] MongoDB, "Install mongodb community edition on ubuntu," web page. [Online]. Available: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

[42] gwillem, "Get ansible to work on bare ubuntu 16.04 without python 2.7," web page. [Online]. Available: https://gist.github.com/gwillem/4ba393dceb55e5ae276a87300f6b8e6f

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. CODE REFERENCES

References used in deployment, benchmarking, visualization programs are formally documented here as well as noted in a comment in the code [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42].

B. EXECUTION INSTRUCTIONS

The project should be run on an Ubuntu 16.04 LTS (Xenial Xerus) machine. The required modules for the project can be installed in a virtualenv virtual environment using the file project/S17-IO-3012/code/requirements.txt.

The main script, project/S17-IO-3012/code/bin/project.py, can be run to execute all functionality. Project.py functions (deploy, kill, benchmark, report, distribution) are described in help, but sample instructions are provided below for each function.

python project.py has four functions.

B.1. deploy
Runs a deployment. Takes 6 parameters:

1. Cloud - chameleon, jetstream, or kilo (futuresystems)

2. MongoDB Version - 34 for version 3.4, 32 for version 3.2

3. Config Server Replication Size - a number

4. Mongos Router Instances - a number

5. Shard Count - a number

6. Shard Replication Size - a number

   Simple example:
   deploy chameleon 34 1 1 1 1
   More complex examples:
   deploy chameleon 32 3 2 3 3
   deploy kilo 34 2 2 1 1

B.2. kill
- Deletes and undefines the current cluster. No parameters.

B.3. benchmark
Runs a benchmark Mongoimport, find, and MapReduce and logs timings. Takes one required parameter - *large* or *small* (for testing purposes).

B.4. report
Regenerates PNG files in the code/report/directory based on current benchmarks

B.5. distribution
Shows the data distribution of the current configuration. For meaningful results, must be run after benchmark.

C. DIRECTORY STRUCTURE

The project/S17-IO-3012/code contains several directories.

C.1. benchmark/
Contains all benchmark timing logs

C.2. bin/
Contains all Bash and Python code

C.3. configfiles/

Contains all configuration file templates

C.4. deploy/

Contains all deployment timing logs

C.5. json/

Contains all json documents

C.6. playbooks/

Contains all Ansible YAML files

C.7. report/

Contains all reports in PNG format

C.8. stdlist/

Contains all bash script output logs

C.9. work/

Contains temporary work files