Spring 2013

MongoDB Performance In The Cloud

Tudor Matei
San Jose State University

MongoDB Performance In The Cloud

A CS298 Project Report

Presented to
The Faculty of the Department
of Computer Science
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Tudor Matei
May 2013

Abstract

Web applications are growing at a staggering rate every day. As web applications keep getting more complex, their data storage requirements tend to grow exponentially. Databases play an important role in the way web applications store their information. Mongodb is a document store database that does not have strict schemas that RDBMs require and can grow horizontally without performance degradation. MongoDB brings possibilities for different storage scenarios and allow the programmers to use the database as a storage that fits their needs, not the other way around. Scaling MongoDB horizontally requires tens to hundreds of servers, making it very difficult to afford this kind of setup on dedicated hardware. By moving the database into the cloud, this opens up a possibility for low cost virtual machine instances at reasonable prices. There are many cloud services to choose from and without testing performance on each one, there is very little information out there. This paper provides benchmarks on the performance of MongoDB in the cloud.

Table of Contents

1. Introduction

Web applications are growing at a staggering rate every day. As web applications keep getting more complex, their data storage requirements tend to grow exponentially. Relational databases are great pieces of software that allow records to be stored and retrieved in table structures. Unfortunately, the tables have limitations on how they can grow horizontally, column wise because of the amount of space it requires to store each row. Taking the data and splitting it up into multiple tables or normalizing solves the issue, however, this means data is now scattered around the disk and requires multiple reads in different sectors of the drive to retrieve the information.

With the data required no longer contained in one spot a database query to retrieve records from tables with millions or billions of rows can easily start to get overwhelming and take longer amounts of time to return results. Hard drives are still a big limitation for I/O access and most databases require huge amounts of space for storing their structures.

There are several advantages and disadvantages for both SQL and NoSQL databases. Developers often wonder which database is better and which they should use on their project. There is no easy or straightforward answer for this question and there is not one database that will work for every project. MongoDB has both strengths [1] and weaknesses [2], but overall it performs quite well and it doesn't have as many restrictions and limitations as other NoSQL databases [3][4][5].

Twitter is a great example of a company that likes to experiment with NoSQL databases [6]. They use Scribe for logging massive amounts of data, Hadoop for power of map reduce to run analytics on large amount of data using Pig and Hbase, FlockDB as a distributed graph database,

Cassandra as a completely distributed database [7] for atomic operations and many others [8]. This bold move to integrate NoSQL databases resulted from storage limitations and downtime occurring with their MySQL cluster [9].

MongoDB is already being used in production by some of the biggest online companies [10]. Craigslist uses it for storing over 5 billion classifieds records. MTV uses for their CMS, which updates the content for all their major websites. Sourceforge uses the database for their backend storage of: front pages, project pages and download pages. Disney uses it for their games to store a persistent state of their common objects. IGN uses it for real-time traffic analytics and RESTful Content APIs. Electronic Arts uses it for serving game feeds on their website, that are cached up from different components. Shutterfly uses it almost exclusively for all their needs, especially with the storage of photo metadata. Online newspaper websites such as The Guardian, Examiner, Business Insider, Forbes, New York Times, Chicago Tribune all use the database in some way either for their API, comments or blog posts. Intuit uses it as a real-time monitoring tool for how their users interact with their website. Bit.ly uses it the database to store the entire history of all the clicks. FourSquare uses the database to store user check-ins – by combining AWS cloud computing and MongoDB shards [11]. There are many other companies such as github, ShareThis and many others that use MongoDB internally for their own reporting.

2. Background

Different ways of storing data have been a big subject of discussion since the early databases. One of the most popular ways to store information is inside a flat file, containing delimiters that separate fields of data. An example that is still being in use today is the CSV (Comma Separated Values) file, where each field's value is stored inside the file with a comma (delimiter) separating the values.

A disadvantage of such a database structure based on a flat file is that in order to find a record a system would have to iterate through the entire file in order to find the correct record. With any file that has thousands of records and multiple columns it may take a while to go through each row and column to find the information needed. Another big disadvantage of the flat file architecture is having records that require different number of columns. For example, assuming a CSV file has the following fields: first name, last name, home phone, and email address. If there was ever a need to store a second phone number (cell phone) into the database, then a new column would have to be added and each record in the database would need to be updated so that it supports this new structure [11]. This creates a big problem for files with large rows and adds an overhead for every row in order for the file to be valid [1].

2.1 Relational Databases

A solution for this flat file databases started with the relational databases where the schema is designed and managed through a RDBMS (Relational Database Management System). By having the schema designed in a language such as SQL (Structured Query Language) the underlying architecture can be designed and structured in different ways, therefore allowing the

same language to be used to communicate with different databases. One of the best optimizations that all RDBMS provide are indexes, which greatly speed up a database performance so that the system no longer has to look at every column/row in order to figure out where the record exists, instead it would be able to look it up quickly. An index inside a database is very similar to the index used for a book. When looking for something specific in a book, one would turn to the index section and find the page number where the search term is located.

Relational Databases are able to go one step further and alleviate duplicate data by setting a relationship between two tables, which allows duplicate information to be stored separately in another table and then referenced inside the current table. For example, assuming a table of products has many products and table of users has a list of all the users in the system. By creating a shopping cart table, we can associate a user and a product and reference them to their appropriate table. There is no need to store the user's information and any details of the product inside the shopping cart table; these tables can be accessed using a join or union statement using SQL queries.

2.2 Document Oriented (NoSQL) Databases

Document oriented databases often do not use SQL, but instead it uses a different language to communicate [12]. A document can have any number of fields listed in any order just like a relational database, but unlike relational databases, a row inside a document oriented database does not need to have the same number of types of fields as any other row inside the database. This is due to the fact that there are no schemas that restrict a row to be identical in number and the sequence of fields. While there are many document store databases, MongoDB stands out due to its high performance and ease of setup [13][14].

3. MongoDB

3.1 Document-Oriented Storage

Mongo uses BSON, which is the binary-encoded serialization of JSON format to query and store data [15]. The developers chose BSON instead of JSON because of its efficiency – binary strings can be smaller when encoded, and support of additional data types. JSON currently only supports the following data types: string, number, Boolean, array and object. BSON supports: string, int, double, Boolean, date, bytearray, object, array and others. BSON's only restriction is that data must be serialized in little-endian format. Also, since BSON is the format that the data is sent/retrieved and stored, there is little need to decode it to text.

A database in Mongodb is analogous with a relational database, or a grouping of collections. A collection is a set of documents, similar to a table in a relational database. A document is the most basic entity where Mongodb stores information, similar to a row inside a relational database, except the data structure is schema-less. Perhaps, the most important benefit of a document is that it may contain other documents embedded inside. Below is an example of a document with two embedded documents:

```
{
	"name" : "John Smith",
	"phone" : {
		"home" : "555-123-4567",
		"cell" : "555-321-7654"
	},
	"address" : {
		"street" : "123 Main Street",
		"city" : "San Francisco",
		"state" : "CA",
		"zip" : 94115
	}
}
```

The ability to embed documents is a very powerful concept that is implemented very well inside Mongodb [5]. In order to achieve this type of structure in a relational database, a table would require the following schema: name, phone_home, phone_cell, address_street, address_city, address_state, address_zip. An alternative approach would be to separate phone numbers and addresses into their own separate tables and use two join statements. The advantage of Mongodb is that the entire object could be retrieved into memory using only one disk seek, while a relational database may require up to three disk seeks in order to retrieve the same data.

Although MongoDB does not support SQL statements, it does provide a very easy to use syntax for finding documents using simple JSON syntax. For example, retrieving documents that have the name John Smith can easily be done with the following query:

```
db.users.find( { "name" : "John Smith" } )
```

Alternatively using the same structure of multiple embedded documents, it is just as simple to find users who live in San Francisco.

```
db.users.find( { "address.city" : "San Francisco" } )
```

3.2 Index Support

When tables grow in size, indexes will dramatically speed up queries. Indexes in Mongodb work almost identically with the relational databases. Mongodb indexes are implemented as B-Tree just like the default in MySQL. Indexes can be added to any key including documents and their keys as shown below.

Adding a simple index:

```
db.users.ensureIndex( { "name" : 1 } )
```

Adding an index to embedded keys:

```
db.users.ensureIndex( { "address.city" : 1 } )
```

Adding an index to an entire document:

```
db.users.ensureIndex( { "address" : 1 } )
```

In addition, Mongodb also supports two-dimensional geospatial indexing, with future support for spherical indexing. Two-dimensional geospatial indexing is used for finding places near a specific x, y coordinate. This index is extremely powerful when dealing with location based services [16].

3.3 Replication

Replication is a very important concept for Mongodb and is always the ideal setup for production environments. MongoDB's recommendation of replication was set because prior to version 2.0 there was no single server durability and with a combination of writes without

acknowledgement, there was no guarantee that the data ever got stored [17]. Currently, on 64 bit machines, journaling is enabled by default. With this option enabled, the database provides write ahead logging to guarantee that write operations will occur even if the server crashes [16].

Replication is a concept that duplicates data over a group of servers known as replica sets. In a replica set there is a primary server and one or more secondary servers. With this setup the primary acts as a master, where all writes will go. Secondaries can be used for reading as long as the developer is aware that data will eventually get there. Secondaries are great places to run backups, since it will not increase the load on the primary node. MongoDB can be set up with automatic failover so if the primary node out of the two (or more) nodes goes down, one of the secondaries will step up and become primary in order for writes to continue [18].

3.4 Auto-Sharding

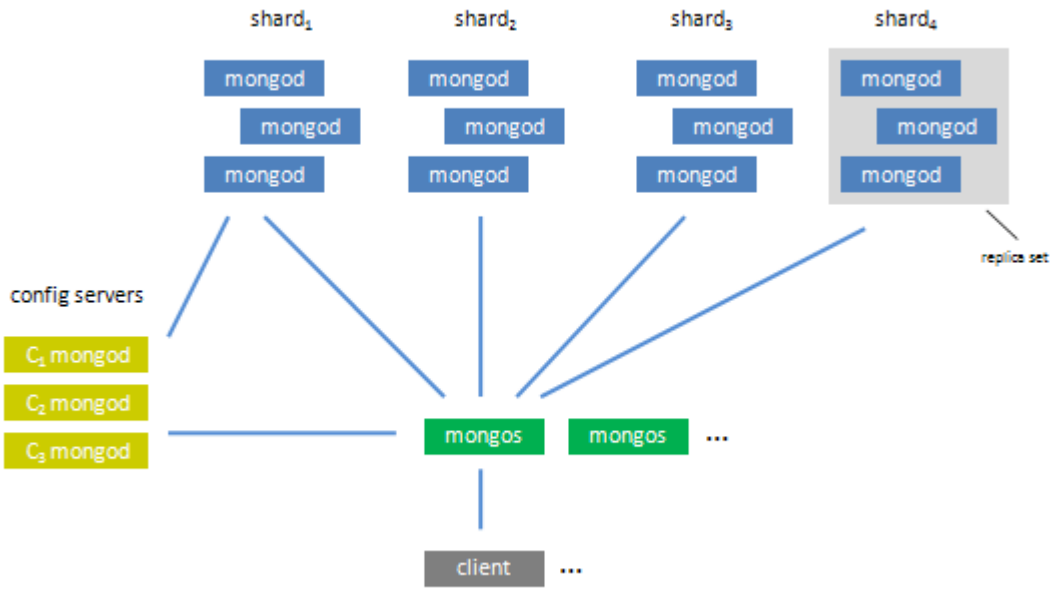

Figure 1 - Sharding architecture of MongoDB [19]

Sharding is a technique that allows records inside a database to be split across multiple servers; this is opposite of normalization which is split by columns into other tables. The process of splitting the rows allows certain ranges of records to be stored separately on different servers. While it is a powerful feature to have, this technology brings up an important issue; what happens when the server that has the data goes offline? Fortunately, auto-sharding with automatic failover is meant to alleviate this problem by automatically switching incoming queries to be routed to the new master replica server and avoid any failover or downtime [20].

The figure depicts the Auto-Sharding process. Data is split up into multiple shards that live on separate servers. For example, data could be split up using ranges of ids or last names. Each shard is running a mongod process and ideally should have at least one replica server per shard, in order to make sure in case of failure that the data is accessible. Config servers are simple mongod interfaces that store the configuration of the shard ranges. The data is replicated between the config servers if there is more than one; in production it's advised to run at least three replica servers. Lastly, the mongos process is responsible for querying the config server and gathering the data from the appropriate shards. This process is lightweight and can be easily run on the application server. The benefit of running it on the application server is that application servers only have to make local requests and the mongos instance takes care of the rest [16].

The minimum requirements for Auto-Sharding to work are at least one mongod config, one instance of mongos and two mongod shards. With this minimum requirement the data is highly distributed and no replica servers. The minimum requirement is at least one replica per mongod shard and each application server should contain one config mongod instance and one mongos instance. With this setup the lightweight clients are working on each application server and keeps the servers down to a minimum while there is at least one backup for each instance [16].

4. Cloud Computing

Within the past few years, the Internet has gone from most websites being served from dedicated hardware boxes to what is now referred to cloud computing or hosted on cloud servers. These servers are not bound to hardware; they are virtualized instances that run on top of dedicated hardware [21][22]. The key benefit of cloud computing is it's elastic ability to create new server instances on the fly, instead of taking the time to buy individual servers and setting them up. This type of technology opened the doors for many cloud providers to offer services that allow on demand cloud servers to be created and paid by the hour [18].

4.1 Amazon

The Amazon Elastic Compute Cloud (EC2), part of the Amazon Web Services (AWS), allows for virtual servers of many different sizes to be created on demand. Amazon changed way people see cloud computing and on demand computing resources. EC2 offers fast creation of computing resources on demand whenever needed and running for as long as needed, with reasonable pricing plans, where customers get charged by the hour. Their wizard allows from a wide variety of operating systems pre-setup and ready for deployment, both provided by Amazon and community based.

4.1.1 Amazon Web Services Architecture

The EC2 allows the customer to pick the type of instance based on different CPU / Memory / Hard Drive specs to fit the customer's needs. EC2 offers CPU power in in the form of compute units or ECU and is equivalent to the "capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon

processor"[23]. Amazon also offers two types of storage options: instance store and EBS. Instance store offers high capacity volatile storage that gets reset upon termination of the EC2. Elastic Block Storage (EBS) is a separate AWS service providing high availability and reliability [24][25]. Amazon claims that EBS instances are 10 times more reliable than hard drive storage. EBS storage is a network attached storage volume that is charged currently at $0.10 per gigabyte-month, and $0.10 per million I/O requests. EBS storages are ideal for databases due to its high reliability as each volume automatically replicated within the same zone [26].

For improved I/O performance, Amazon offers EBS optimized instances, offering higher throughput for provisioned EBS volumes [27]. A provisioned EBS volume is a mountable volume that is guaranteed to perform within 90% of the speed advertised. The guaranteed speed starts from 100 IOPS and goes all the way up to 2000 IOPS with a 10:1 GB to IOPS requirement. Therefore wishing to create a 2000 IOPS volume requires at least a 200 GB volume. There is an extra cost to running an instance that is EBS optimized, currently $0.025 per hour or $0.05 per hour more. The pricing structure for EBS provisioned storage is higher. The price per GB is just slightly higher at $0.125 per gigabyte-month, while the cost per request is much higher at $0.10 per previsioned IOPS month. For example a 200GB volume would cost $43.15 for standard EBS, while it would cost $221.80 for the same size 2000 IOPS provisioned volume [28].

4.1.2 EC2 instances

The smallest and most inexpensive instance they offer is the micro instance with 613 MiB of memory, up to 2 EC units (not guaranteed), EBS storage only and low IO performance [23]. This instance provides a very inexpensive way of testing a web application or database. This instance provides the least amount of performance and is offered as the EC2 instance for the amazon free

trial. At the time of this writing, for the Oregon Zone, the instance costs $0.02 per hour. Since the only way to set this up is via EBS, the instance can easily be shut down and a more powerful one can be started using the same EBS volume [29].

The standard small instance comes with 1.7 GiB of memory, 1 ECU using one virtual core, offering up to 160GB of instance storage or a choice of EBS with moderate I/O performance. There is no EBS optimization available for this instance. The current cost of this instance for the Oregon Zone is $0.12 per hour. Starting with the small instance, there are two types of

The standard medium instance comes with 3.75 GiB of memory, 2 ECU using one virtual core, offering up to 410GB of instance storage or a choice of EBS with moderate I/O performance. There is EBS optimization available for this instance. The current cost of this instance for the Oregon Zone is $0.24 per hour.

Amazon also offers High I/O Quadruple Extra Large Instance with 60.5GiB of memory, 35 ECU using 16 virtual cores, 2 SSD volumes with 1TB storage and very high I/O performance. There is no EBS optimization available for this device since this it's meant to be used with the high I/O storage instance. The current cost of this instance for the Oregon Zone is $3.10 per hour. This instance is currently second to the highest cost and promises to deliver more than 120,000 IOPS random reads and between 10,000 IOPS and 85,000 IOPS for random writes [23].

4.2 Linode

Linode is a hosting company that offers virtual private servers (VPS). Their architecture is based on Xen and is more expensive than the amazon counter part; it offers very good I/O performance for even the low-end instances. Their pricing structure is done per month instead of per hour like

the other competitors, but a customer could cancel their VPS and only get charged up to the nearest day. Linode refers to their Xen virtual private servers as Linodes. What also makes Linode service unique is unlimited inbound transfer and high limits for outbound transfers included. All other services charge extra for inbound and/or outbound transfer [30].

4.2.1 Linodes

Each Linode has access to a quad core CPU, but has different priorities to how much CPU time it can use [30]. Their fair share model limits the number of cycles only if the other instances on the same box require access, otherwise if the instances are idle and instance could use all four cores at full speed. Their lowest Linode has 512MB of memory with access to 24GB of storage, 1x priority for the CPU and 2TB outbound transfer included for $19.95 per month or roughly $0.027 per hour. This research paper won't use the 1GB Linode. The 2GB Linode has access to 48GB of storage, 4x priority for the CPU and 4TB of outbound transfer for $79.95 per month or roughly $0.11 per hour. The 4GB Linode has 192GB of storage, 8x priority for the CPU and 16TB of outbound transfer for $159.95 per month or roughly $0.219 per hour.

4.3 Rackspace

Rackspace cloud allows a customer to sign up for their service and create cloud servers in a matter of minutes. The signup process goes through an in depth verification to make sure that the customer is who they say they are before service is enabled. Rackspace uses OpenStack to manage virtual instances, block storage and usage reporting.

4.3.1 OpenStack

OpenStack, an open source technology developed by Rackspace and NASA to create massively scalable clouds. There is a main dashboard that provides a user interface to create and manage instances, volumes, and networking. The dashboard also gives users the power to check usage, graphs and billing. OpenStack is split into three main components: compute, networking and storage [31].

The compute component is responsible for provisioning and managing large networks of virtual machines. This component encourages horizontal scaling of virtual servers from a shared pool of resources. Each virtual instance can be set up to run under a KVM or Xen hypervisor technology. The compute component is written in Python, which enforces portability and even being able to run on ARM architecture.

The storage component is responsible for the object and block storage of data. The object storage provides redundant and distributed file storage with no central point of failure, allowing commodity hard drives to be used to achieve storage capabilities in terms of petabytes. Files are automatically replicated across multiple hard drives in order to ensure protection against drive failures. Block storage on the other hand is used for applications that require raw disk access, such as an operating system or database.

The networking component is responsible for separating the outside network from the internal. Networking and security for a large cluster of virtual instances can become overwhelming if managed manually. This component is meant to handle static IP / DHCP allocation, load balancing, firewall, and VPN.

4.3.2 Cloud Servers

There are several cloud server templates that Rackspace offers. Rackspace puts limits on both the internal and public network speeds, but they are not relevant in this research paper [32]. The smallest instance is 512MB memory with one virtual CPU, 20 GB of disk space for $0.142 per hour. There is also a 1GB memory instance that offers 40GB of disk space, but for this research there is no reason for testing the 1GB instance since amazon does not offer anything equivalent. The 2GB memory instance offers 2 virtual CPUs with 80 GB of disk space for $0.24 per hour, and the 4GB memory instance offers 2 virtual CPUs with 160GB of disk space for $0.36 per hour.

4.4 Windows Azure

Windows Azure is the newest member to the cloud servers [33]. Their virtual servers instances in preview mode with lower than average pricing; their pricing model may change after the beta period is over. Unlike other providers, the prices are the same if a customer chooses a Windows or a Linux Virtual Machine. This is important if the application stack is a mix of both environments [34].

4.4.1 Virtual Machines

Windows Azure provides both Linux and Windows instances starting with the extra small instance with 768MB of memory and a 1GHz CPU for $0.013 per hour. The small instance running at 1.75GB of memory and 1.8GHz CPU costs $0.08 per hour. Their medium virtual instance has access to 3.5GB of memory with 2 CPU cores running at 1.6GHz for $0.16 per

hour. Creating a virtual machine instance in Windows Azure provisions a virtual instance and a storage node with 30GB of storage by default. Extra storage can be added anytime to an instance.

5. Implementation

Each virtual instance will have installed MongoDB, PyMongo and a benchmark python script. MongoDB latest version 2.2.3 will be used for all these tests with the default setup configurations. PyMongo latest version of 2.4.2 is installed via easy_install command. All servers use a variation of the CentOS or Fedora operating systems. The python script gets copied over to the server when the appropriate fabric script command is executed.

5.1 Fabric

Fabric is a python library and command line application that eases deployment of commands to a remote server [35]. By utilizing ssh keys, fabric can easily manage a wealth of logical commands that would be more difficult to achieve using bash script. Fabric is written in python and executes given tasks on remote servers. The main benefit of using fabric is that a task could be executed on multiple servers in parallel. Benchmarking can take a long time to complete, so by running these commands in parallel across multiple hosts, one could get results much faster. There are three main tasks that are executed for any new server instance.

The first task is the called ‘push_ssh_key’ and is responsible for making sure that the ssh key is added to the user that is in charge of running everything. The user can be anybody as long as they have access to sudo for the next step. The script pushes the main public key in ~/.ssh/id_rsa.pub to the server and makes sure that it doesn’t get written twice. By adding an ssh key, logging in to the server is much easier and password does not have to be supplied multiple times.

The second task is named ‘install_mongodb’ and is responsible for installing MongoDB. The command will install the latest version all the time, because it is set up to look in the 10gen RPM repository. This task will also install python-setup tools, which is needed to install PyMongo, the driver that communicates between python and MongoDB. This task is also responsible for setting up the database daemon to start right then and at startup.

The third task is named ‘mongodb_perf_all_remote_tests’ and is responsible for running all the test on the remote server. This task was originally designed to run each test individually and gather the output locally, but the server’s ssh connection would often time out when running one of the tests that took over an hour to finish. In order to get around this issue, the task now copies the benchmark python script and a bash script (used to executes all the benchmarks in order), locally on the remote machine and runs it using nohup so that if the ssh connection gets terminated then it will continue until the script finishes.

5.2 Benchmark Script

The benchmark python script is responsible for testing writes or reads in MongoDB. The script can run with several command line options. A hostname, port, database name can be specified if the test is not running on the same server as the database, so by default these are localhost, 27017 and ‘test_database’. There are several variables that can be used when testing writes in MongoDB. Total number of processes can be specified via command line, indicating how many processes to use in parallel to write data. The write concern is a very important argument that can be ‘no-ack’, ‘ack’, ‘ack-with-journal’ or ‘ack-with-fsync’. These represent the MongoDB write concerns mentioned in the beginning of the paper where ‘no-ack’ represents no acknowledgement where the script does not wait for the database to respond, runs in

asynchronous mode. The ‘ack’ implies wait for acknowledgement that the database received the request; this is now the default option in the latest drivers. The ‘ack-with-journal’ option makes the script wait until the database writes the data to the collection as well as the journal. The ‘ack-with-fsync’ option waits for the database to flush the data on the disk before returning; this option has been removed from the test benchmark because it yields the exact same results as ‘ack-with-journal’. A toggle to switch between a write or read benchmark can be set. There is also a flag for verbosity that turns on a progress indicator that outputs every second how many records have been either read or written so far.

In order to prepare for benchmarking writes in MongoDB, the script first clears out any records in the collection. By having an empty collection, each test can start on a clean slate. The documents that get inserted are very small in size and represent a test with quick writes. Considering that MongoDB does not handle documents larger than 16MB, it is not feasible doing large document writes.

For benchmarking writes, the script creates a new MongoDB connection for every process so that connection pooling will not be a determining slowdown factor. By default PyMongo creates a connection pool and uses that for communicating with MongoDB. Unfortunately when the number of processors gets higher than 16, these connection pools get shared and results start becoming skewed. This was especially noticeable when the script was initially written to use threads instead of separate processes and each of the threads was waiting for a connection to be available in order to send its data.

The read benchmark, for simplicity, assumes the same number of documents were inserted as there are being read. The script takes the list of all records and shuffles their ids around so that

they are not read in order; this procedure will force the disk to use random access to get these records.

5.3 Getting Results

Once a server is set up through one of the cloud providers, three fabric commands will run locally in the exact order, replacing <username>, <hostname> and <port> with the right credentials.

```
fab push_ssh_key -H <username>@<hostname>:<port>
fab install_mongodb -H <username>@<hostname>:<port>
fab mongodb_perf_all_remote_tests -H <username>@<hostname>:<port>
```

6. Results

All of the tests recorded how long it took to write / read 100,000 records to MongoDB. Different type of write concerns mentioned above were used as well as multiple processors writing or reading to the same database. The tests start with 1 process and doubles each time until it reaches 128 processes. When running 128 processes, there are 128 connections to MongoDB, individually writing or reading data.

6.1 Extra Small Instance Tests

The tests for the lowest amount of memory are different for every cloud provider. Linode has 512MB instance, Amazon has the micro 613MB instance, Rackspace has the 512MB instance, and Azure has 768MB instance; they are all relatively close to each other, within 256MB over the 512MB instance.

The no acknowledgement test results between Amazon micro EC2 instance and all other services are big. The Linode instance was the quickest to finish at 21.42s to write the 100,000 records, followed by Rackspace at 53.72s, Azure at 54.57s and the EC2 at 261.85s. EC2 is clearly much slower for the micro instance to write records as quickly as possible. As the number of processes increase, most of the instances don't change much however

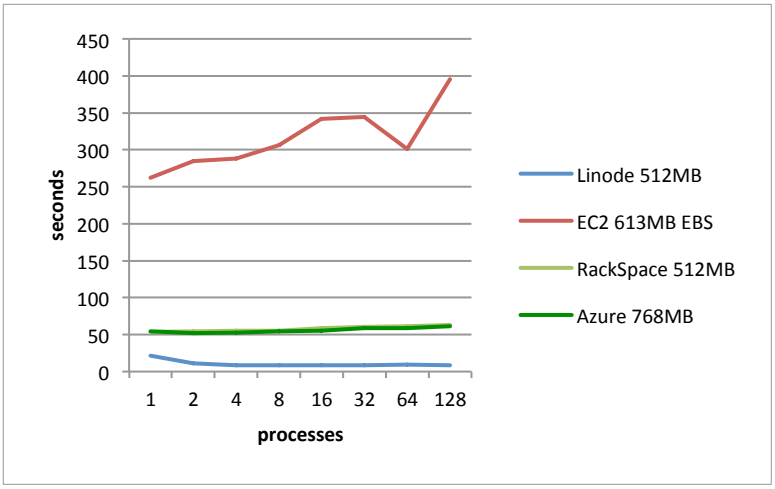

Figure 2 - No Acknowledged Writes for X-Small Instances

the Linode time to complete drops significantly between using 1 process and 4. This is most likely due to its quad core processor that is available for longer periods of time. For 128 processes Linode is still in first place, followed by Azure, Rackspace and then EC2. Azure and Rackspace tests are very close to each other in these tests, so there isn't a clear winner. EC2 on the other hand increases from 261.85s to 395.34s, or a 50% increase in time it took to write the results.

For acknowledged writes, numbers are higher due to the drivers waiting for an ok (or fail) message. Linode started out in first place with 43.78s, followed by Rackspace with 73.07s, Azure with 83.36s and finishing off with Amazon EC2 at a much higher 466.55s. As the number of processors increase, Linode time to complete goes down to 11.54s, quickly dropping down to about 10s at 4 processors. Rackspace and Azure slowly increase from 73.07s and 83.36s to 88.84s and 90.35s respectively. EC2 time to complete goes up 22% from 466.44s to 570.23s.

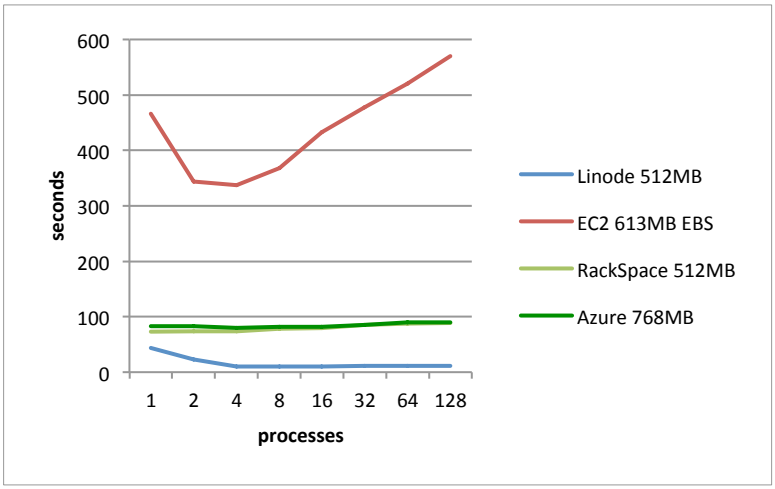

Figure 3 - Acknowledged Writes for X-Small Instances

Journal writes stress test shows the true I/O potential, since each database write call waits until the command is added to the journal log. The finish times are very closely related for 1 process around 3,600s, except for Azure with 4,502.53s (30% difference). As the number of processes increased, the tests start to show their differences. All cloud providers run faster as additional processes are added except for EC2 instances where after 16 processes starts to increase. This shows the true limitation of EC2 micro instances. At 128 processes Linode is still the fastest and finished at 36.91s, followed by Rackspace and Azure close to each other at 104.68s and 105.55s and then followed by EC2 with 731.71s.

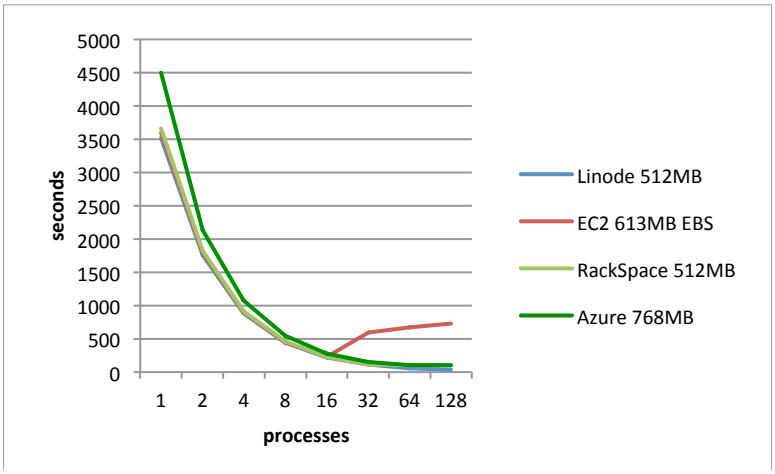

Figure 4 - Journal Writes for X-Small Instances

Reads from a database are also important and for 1 process Linode finishes in 42.62s, followed by Rackspace and Azure at 52.23s and 53.77s. As number of processors increase, the time that it takes to complete the test increases as well for all instances except Linode, which goes down to 10.11s, while Rackspace and Azure go up to 68.34s and 64.35s. As before, Amazon EC2 has much higher completion times at 427.21. It's worth nothing here that Amazon

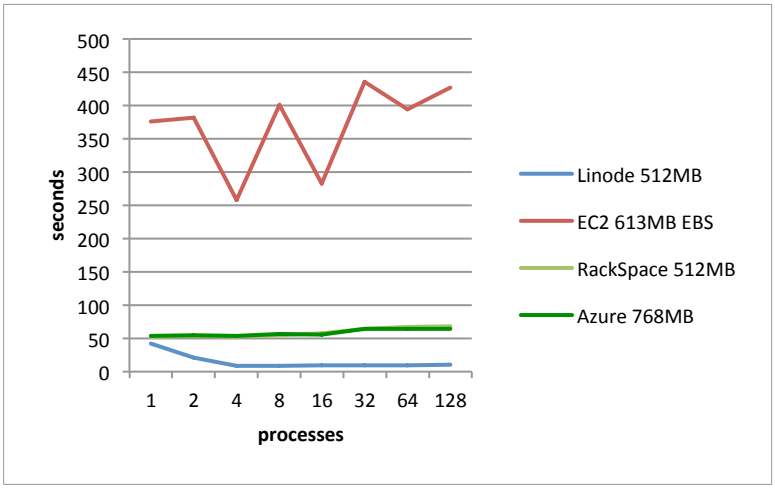

Figure 5 - Reads for X-Small Instances

EC2 has sporadic completion times going from 381.96s with 2 processes, down to 257.94s with 4 processes, back up to 401.11s with 8 processes and back down to 227.35s with 16 processes.

6.2 Small Instance Tests

The tests for the small instances, the amount of memory is also different for every cloud provider. Linode has 2GB instance, Amazon has the micro 1.7GB instance, Rackspace has the 2GB instance, and Azure has 1.7GB instance. Starting with the small instance Amazon EC2 has two options, instance store and EBS, so the tests for EC2 are separated. For the EC2 with instance store, mongo.conf was manually updated to point to the instance store partition since the default root partition is EBS.

For the small instance with no acknowledgement tests, completion times are lower than the extra small instances. For 1 process, once again Linode leads with a completion time of 40.64s, Rackspace and Azure come closely behind at 48.29s and 50.62s.

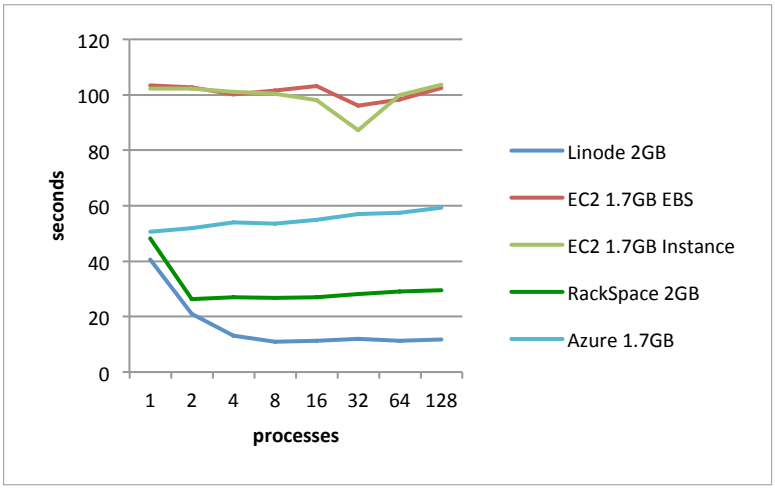

Figure 6 - No Acknowledgement Writes for Small Instances

Amazon EC2 results are much slower again with a small difference between Instance Store (102.28s) and EBS (103.35s). As the number of processes increase, Linode results are quite outstanding, shown to handle concurrent write requests very well. Rackspace got a much better 128-process test result of 29.43s for the small instance, than the 61.92s for the extra small.

MongoDB on Azure and Amazon is struggling to keep up with the inserts; fortunately the completion times are not getting much higher.

Acknowledged write results are very similar to the non-acknowledged results, but slightly higher completion time. For 1 write process Linode is still the fastest to finish at 40.64s seconds, followed by Rackspace at 48.29s, Azure at 50.62s, EC2 Instance Store at 102.28s and EC2 EBS at 103.35s. As the number of processes increase EC2 instances slowly increase in time, Azure stays roughly flat, Rackspace decreases quickly at 2 processes and then stays flat all the way to 128 processes.

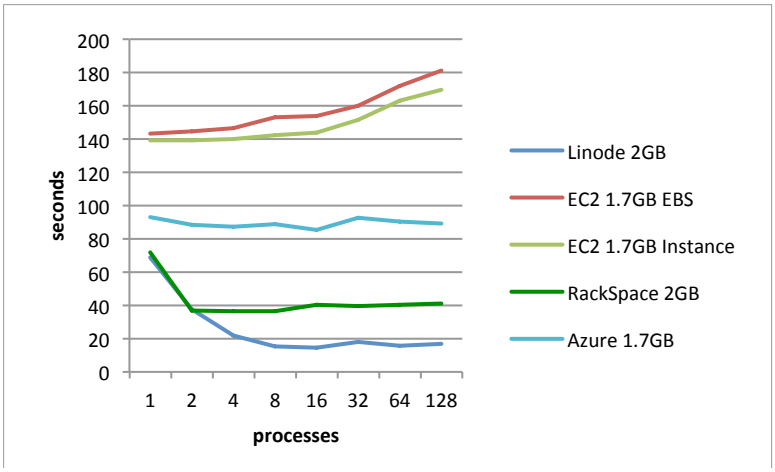

Figure 7 - Acknowledged Writes for Small Instances

Journal writes look less chaotic for small instances than they did for the extra small. This is party due to Amazon EC2's better handling of I/O. From the graph, all services look very close to each other, however this is not the case due to the scale used. For 128-process, results are different enough with Linode leading at 28.11s, Rackspace at 55.03s, Azure at 107.67s, EC2 Instance Store at 191.91s and EC2 EBS at 208.06s.

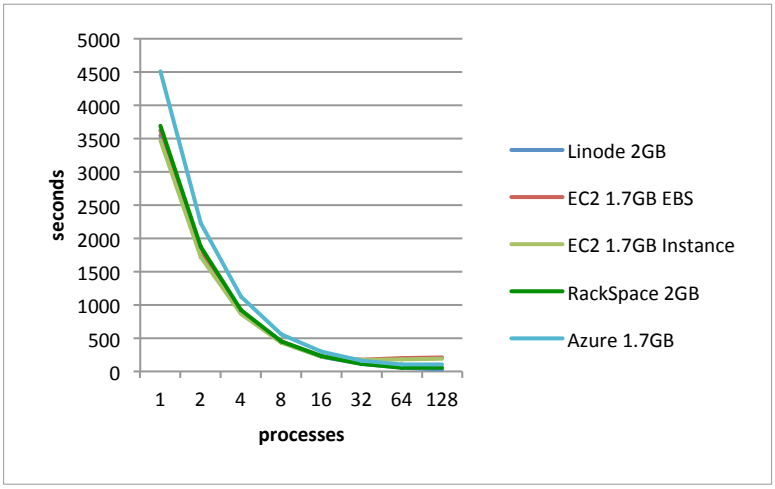

Figure 8 - Journal Writes for Small Instances

Read results are unfortunately still higher than expected for EC2 instances. Results for EC2 are much more stable for the small instances than the extra small instance, where they were all over the place. With Linode leading in final results, Rackspace managed to get better results for 2-process reads with a slightly lower test completion time at 26.36s vs. Linode at 29.62s.

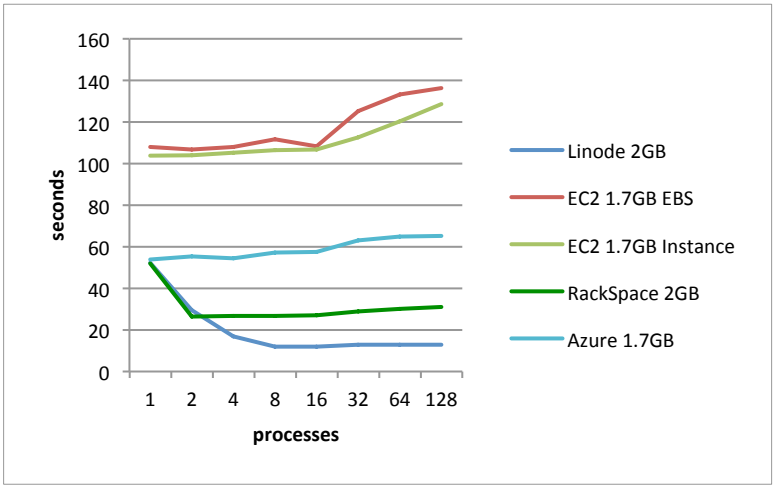

Figure 9 - Reads for Small Instances

The results show that for Amazon EC2 instances, the instance store is slightly faster than the EBS store, however the benefits of EBS outweigh this small benefit. The EBS has more benefits, especially being able to last between instances. If the instance is running too slowly, it can easily be upgraded without having to reinstall anything.

6.3 Medium Instance Tests

The tests for the medium instances, the amount of memory is also different for every cloud provider. Linode has 4GB instance, Amazon has the micro 3.75GB instance, Rackspace has the 4GB instance, and Azure has 3.5GB instance. For the Amazon EC2 tests, there is an additional EBS test with a RAID-10 setup via Linux raid. The setup involved creating a total of four EBS volumes with two of them set to stripe and the remaining two mirroring.

For non-acknowledged 1-process test results Linode was not the fastest, finishing in 62.67s being the slowest of all cloud providers, followed by EC2 EBS finishing in 51.93s, EC2 EBS RAID 10 in 51.68s, Rackspace in 44.58s and Azure in 38.78s. After increasing the number of processes to 128, Linode was still the fastest at 15.14s, followed by Azure in 26.58s, Rackspace in 29.66s, EC2 RAID 10 in 58.87s. There is only a slight improvement between the EC2 EBs and RAID10 EBS.

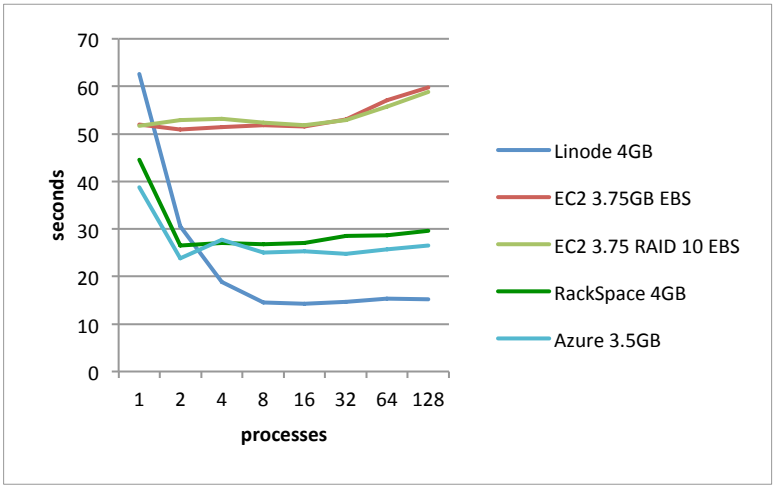

Figure 10 - No Acknowledge Writes for Medium Instances

For acknowledged tests, the results are very similar to non-acknowledged, but the Azure instance performed the worst, finishing in 99.23s, followed by Linode instance in 93.51s, EC2 EBS in 73.83s, EC2 EBS RAID 10 in 72.97s and Rackspace in 71.12s. For the 128-process test, Linode performed the best, finishing the test in 21.61s, followed by Rackspace in 41.14s, then Azure closely behind in 44.88s and the two EC2 instances EBS RAID 10 in 84.82s and EBS in 83.64s.

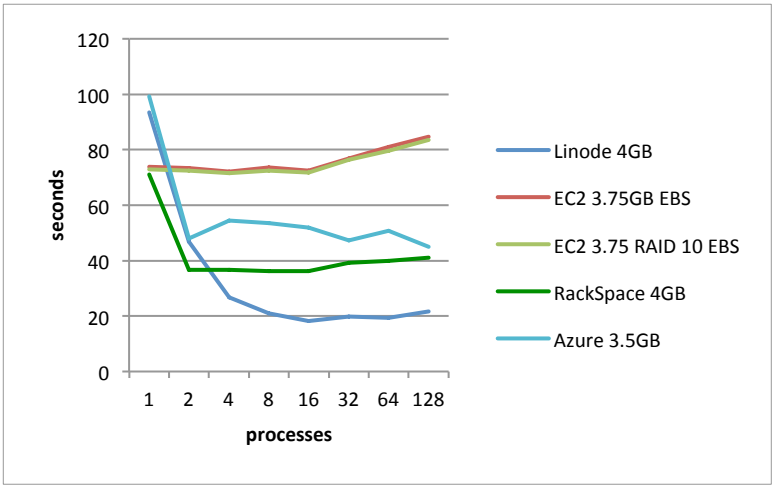

Figure 11 - Acknowledged Writes for Medium Instances

Journal writes for the medium instances start out very similarly to the small instances. At 128-process Linode finished in 28.83s, Rackspace in 52.83, Azure in 59.94s, EC2 RAID 10 in 100.29s, and EC2 EBS in 101.21s. The medium normal EC2 EBS instance finished the journal writes twice as the small EC2 EBS instance.

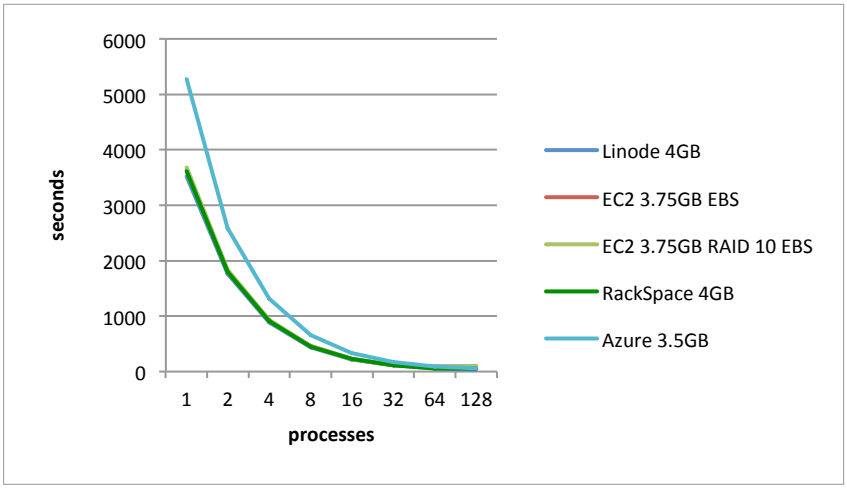

Figure 12 - Journal Writes for Medium Instances

For the 1 process read tests, Linode is once again lagging behind in these series of tests. Linode finished the test in 64.71s, EC2 RAID 10 in 53.29s, EC2 EBS in 51.83s, Rackspace in 51.52s and Azure in 50.36s. With more processes Linode's completion time dropped significantly to 15.25s, Azure and Rackspace close to each other 29.49s and 30.93s. Performance of EC2 with RAID 10 dropped down to 61.12s and ECB in 62.81s.

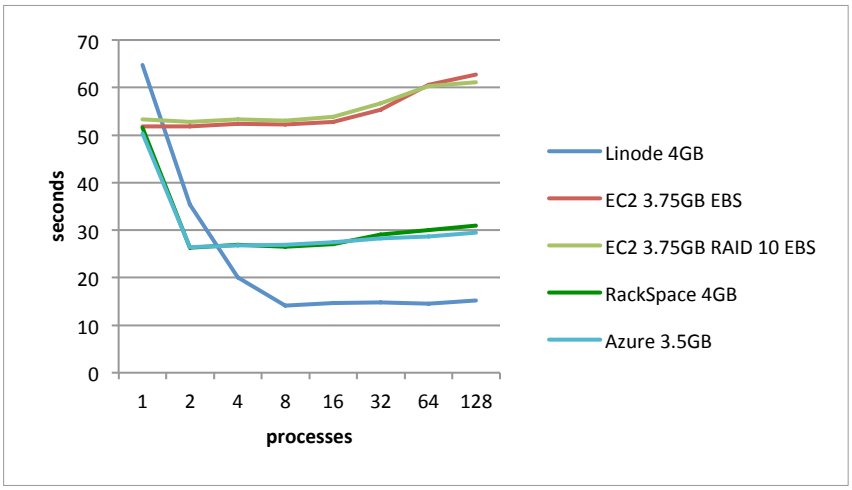

Figure 13 - Reads for Medium Instances

6.4 High I/O Performance Tests

In order to understand the performance of cloud / virtual servers, research with a non-cloud computer must be done. The computer is an Apple MacBook Pro with a Quad Core 2.3GHz Intel Core i7 processor, 16 GB of 1600MHz DDR3 memory and a Kingston SH103S3240G, 240GB Solid State Drive. The SSD is a consumer hard drive that is advertised to be able to reach sequential reads speeds of up to 555MB/s, sequential writes up to 510MB/s. The drive is also able to reach up to 86,000 IOPS for random reads and 60,000 IOPS for random writes, with sustained random writes of 40,000 IOPS and sustained writes of 57,000 IOPS.

For the high performance tests, MongoDB was tested on the MacBook Pro (SSD), EC2 Large EBS Optimized (non-SSD), EC2 High I/O instance (SSD) and the results from the previous tests were used for Linode 4GB due to the fact that I/O performance won't improve by going any higher, since I/O is equally guaranteed to all Linodes.

For non-acknowledged writes, the performance is stable between all the tests. With one processor, the completion times are 12.61s for MacBook Pro, 37.56s for EC2 High I/O SSD, 40.91s for the EC2 EBS Optimized and 62.67s for Linode. As time progresses, unlike previous tests, the instances start to flatten out and give constant results. At 128 processors, EC2 High I/O SSD completed the

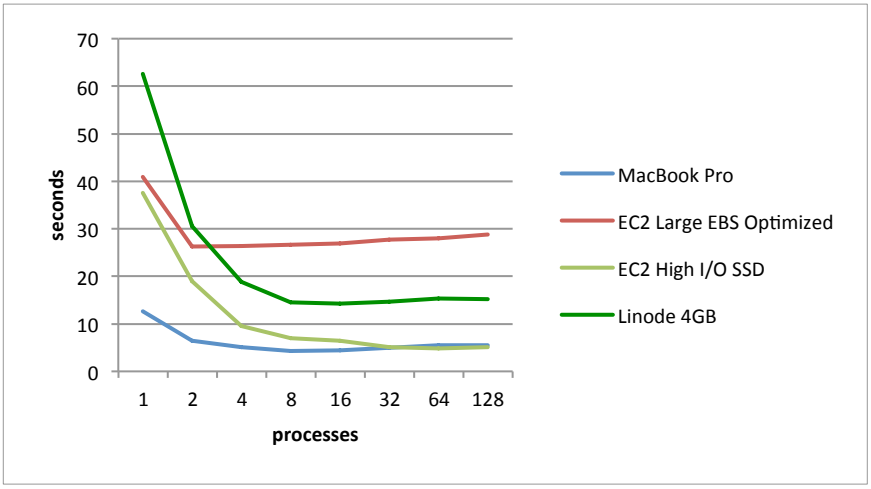

Figure 14 - No Acknowledge Writes for High Performance Instances

test in 5.06s, MacBook Pro in 5.56s, Linode 4GB in 15.14s and EC2 Large EBS Optimized in 28.81s. It can be seen clearly that the EC2 High I/O are using enterprise grade SSD with higher performance than the consumer one used in the laptop.

For acknowledged writes, the tests are very similar to the non-acknowledged with MacBook Pro finishing in 23.48s, followed by EC2 High I/O SSD in 70.19s, EC2 Large EBS in 70.91s and Linode 4GB in 93.51s. After settling down at 128 processors the test results are as follows: MacBook Pro finished in 7.08s, EC2 High I/O in 10.89s, Linode 4GB in 21.61s and EC2 Large EBS Optimized in 41.24s. The results of the SSD on the Mac were very close to the ones on the EC2.

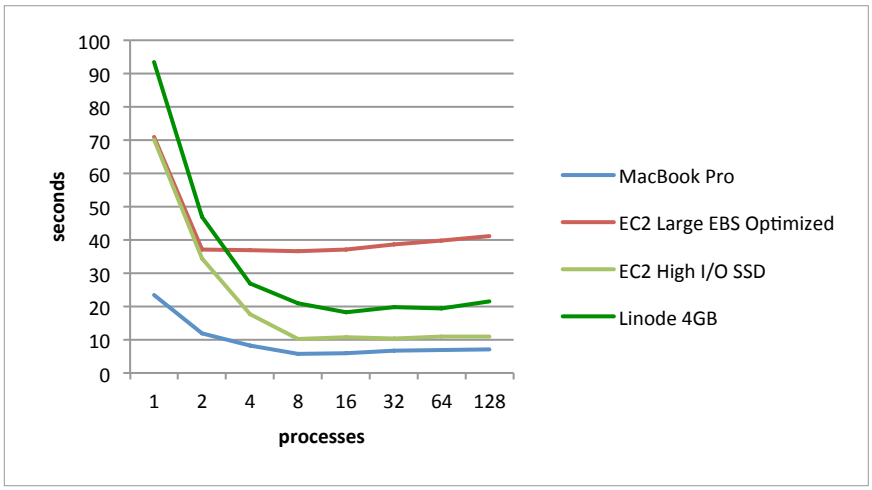

Figure 15 - Acknowledged Writes for High Performance Instances

For journal writes, results are very close together for the most part: EC2 High I/O finished in 3479.26s, MacBook Pro finished the test in 3522.79s, Linode in 3523.94s and EC2 Large EBS in 3601.62s. The values are so close to each other that the differences are negligible. Finishing off with 128

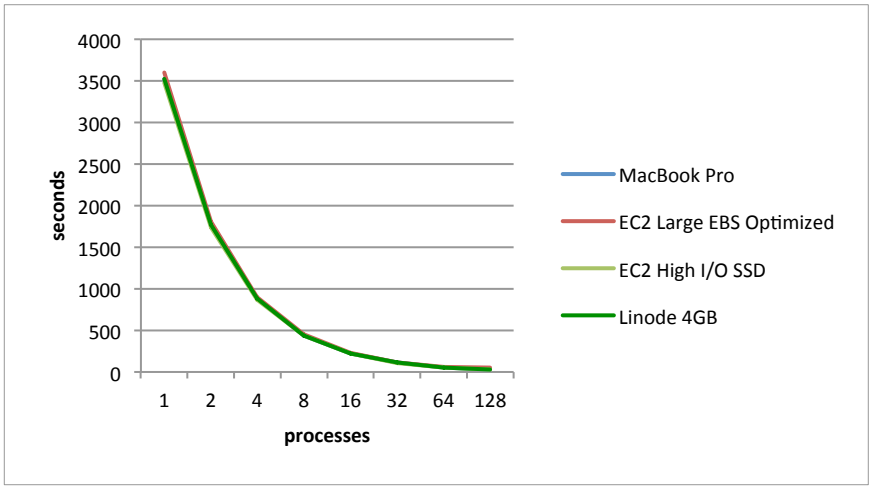

Figure 16 - Journal Writes for High Performance Instances

processors, EC2 High I/O SSD finished the test in 27.62s, followed by MacBook Pro in 28.39s, Linode in 28.83s and EC2 Large EBS in 49.81s. Values are all very close, even the EBS is able to keep up with the demanded throughput.

The read results for these high performance instances are very fast. With 1 process, MacBook Pro finished the test in 40.92s, EC2 Large EBS Optimized in 50.62s, EC2 High I/O in 52.17s and Linode 4GB in 64.71s. At it's maximum throughput using 128 processes, EC2 High I/O finished in 5.82s, followed by MacBook Pro in 9.69s, Linode in 15.25s and EC2 Large EBS Optimized in 30.02s.

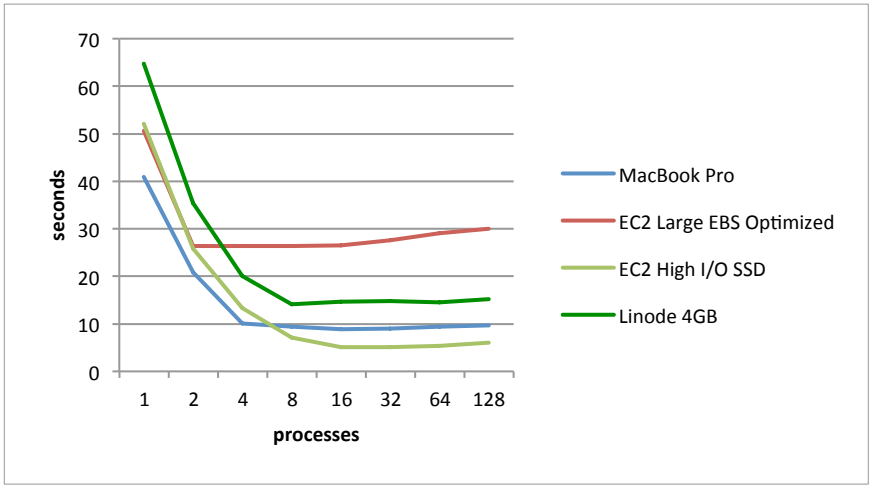

Figure 17 - Reads for High Performance Instances

7. Recommendations

The results clearly show that equivalent instances between providers have completely different MongoDB performance metrics between the cloud providers and their plans.

Linode is a cloud service provider that has no limits on I/O for their virtual instances, no matter the size. A 512 MB Linode will perform just as fast as the 4GB Linode when reading and writing MongoDB data to the storage. This is clearly shown in the numerous tests using journal writes and simple reads. Although the time to complete the tests has gone up as the Linode levels increased, the machines are added to different arrays of storage, so variance could exists. Even with the increase in time to complete the results, these virtual instances still performed much better than the other providers.

Rackspace, with their OpenStack cloud computing, offers fast speeds for all their cloud instances. Results show an overall tie in performance of Rackspace and Azure, except in some tests where Rackspace performed better; more specifically for all tests done on the Rackspace 2GB instance performed the same as Rackspace 4GB instance. The tests show that Rackspace probably is maxed in term of I/O performance at 2GB. It seems that upgrading to 4GB is not needed unless the customer requires more space or higher public / internal speeds. The same number of vCPUs is used for both instances.

Windows Azure has their own architecture for creating virtual instances. It is still currently in beta mode, so a customer would have to sign up for their program before creating a new instance. Like mentioned above, the performance of MongoDB in an Azure instance is roughly equivalent to the Rackspace instance.

Amazon EC2 equivalent instances performed the worst. It's well known that Amazon EC2 does not have the fastest I/O performance, however the access to EBS resources is very much limited. This limitation comes from limited IOPS put in place on the EBS, which is up to 100 IOPS. Provisioned EBS option is available but only for larger instances. It's not recommended to run MongoDB on any anything smaller than the large instance with optimized EBS. As shown in the results, EC2 instances have very inconsistent test results especially for the micro instance.

With the recent release of the EC2 High I/O SSD instance, Amazon is able to keep up with the rest of the cloud providers. As shown in the test results, this type of instance performs faster than the Linodes do. This of course comes with a high cost, as well as no EBS storage. These instances only provide internal storage at high I/O throughput. In terms of cost, running such an instance will set customers back quite a bit, which is one of the most expensive instances that Amazon offers. MongoDB can be safely ran from these instances as long as they are not terminated and data is replicated onto other instances that run with EBS.

Cost comes into place and Linode 4GB costs $0.22 per hour is a great starting price for high performance MongoDB instance. Similar instance hosted with Rackspace costs $0.24 per hour, Azure costs $0.16 per hour and Amazon costs $0.12 per hour. Although Amazon has the lowest price for the same amount of ram, the performance is nowhere near the competition.

Linode has by far the best performance per dollar. The highest cost instance from Linode is $1.11 per hour, significantly better priced than the Amazon High I/O SSD instance, which costs $3.10 per hour.

8. Conclusion

MongoDB performance can vary significantly between cloud providers due to many different factors. The most important factor for MongoDB is the ability to cache all the indexes in RAM and fast I/O access. The database does not require high computing performance, so it can handle multiple threads without being overloaded. Tests of up to 128 concurrent processes trying to write data at the same time did not bring down any instance, including the smallest instances such as Amazon with a shared CPU core.

Test results show Linode virtual instances have the highest MongoDB write and read performance. These virtual instances also have the best performance for the price per hour. Linodes do not have any caps on disk throughput therefore a 512MB instance will have the same I/O performance as a 4GB instance. This unique feature allows a startup to scale up in memory and disk space with knowing that the new instance will receive the same disk throughput.

MongoDB read and write performance for Rackspace and Windows Azure cloud instances had relatively equal test results to each other. As the instance size increased, performance increases with it. Windows Azure instances steadily performed better with each iteration. Rackspace has no distinction between the 2GB and 4GB instance.

MongoDB running on Amazon EC2 had very poor performance results for the equivalent instance size as the other cloud providers. In some cases, for the lower instance sizes, the performance decreases significantly, between 20 and 50 percent, as more concurrent processes are used. Amazon does offer competitive packages at higher costs, but they come at a much higher cost.

Amazon is currently the number one cloud provider [24]. Amazon focuses their advertising towards creating on demand instances quickly through the console interface or API. Most customers compare instances according to the amount of memory provided; because of this, Amazon EC2 excels, they offer high memory instances at affordable prices. Unfortunately, EC2 CPU speed and I/O performance falls behind the competition. The biggest reason why Amazon has such poor I/O performance has to do with the EBS storage not being local to the virtual machine. EBS is a distributed Network Attached Storage (NAS) with data duplication to ensure the guaranteed durability.

The provisioned EBS volumes are meant to increase I/O performance by paying extra for more IOPS. A customer would expect that provisioning a volume for a specific number of IOPS would result with the same I/O performance no matter which instance was chosen, however this is not the case. It is very important to choose an instance that is EBS optimized in order to take advantage of the I/O performance increase, which is offered by these provisioned volumes. Amazon currently has two different levels of EBS optimized instances: 500Mbps and 1000 Mbps. The smallest instance to offer this option is the large 7.5GB instance.

The EC2 High I/O SSD instance does not provide EBS storage by default, because it would be impossible for an internal network to handle up the SSD read/write speeds of 85,000 IOPS. These instances use two instance store volumes, which do not go over the network and are attached to the instance itself to provide maximum performance. Since the instance storage is volatile, the customer is now responsible for backing up the data and having a failover plan in case the instance is terminated accidentally or for an upgrade.

9. Future Work

Real world testing is very important. The current tests can be further expanded to create MongoDB documents of different sizes ranging from a few KB to the maximum document size of 16MB. The newest trend is to store as much data as possible and later figure out what to do with it. MongoDB offers MapReduce written in JavaScript that allows users to search through massive amount of data. Sharding in MongoDB allows horizontal scaling and testing multiple server setup with storage and retrieval of data is very important for real world applications.

An important factor not considered in these tests is the performance of replication between MongoDB instances. Rackspace as well as Amazon limits the network speeds for their instances. Further experimentation with sharding would be useful in determining how well each cloud provider handles communication within the network and between data centers.

10. References

[1] B. Wong, “Choosing a non-relational database; why we migrated from MySQL to MongoDB,” *Online*, 2010. [Online]. Available: http://benjaminwss.posterous.com/choosing-a-non-relational-database-why-we-mig/.

[2] E. Gunderson, “Two reasons you souldn’t use MongoDB,” *Online*, 2010. [Online]. Available: http://ethangunderson.com/blog/two-reasons-to-not-use-mongodb/.

[3] N. Leavitt, “Will NoSQL Databases Live Up to Their Promise?,” *Computer*, vol. 43, no. 2, pp. 12–14, Feb. 2010.

[4] D. Mytton, “Choosing a non-relational database; why we migrated from MySQL to MongoDB Folllow Up.,” *Online*, 2010. [Online]. Available: http://blog.boxedice.com/2009/07/25/choosing-a-non-relational-database-why-we-migrated-from-mysql-to-mongodb/.

[5] R. M. Lerner, “At the forge: NoSQL? I’d prefer SomeSQL,” *Linux Journal*, p. 192, 2010.

[6] E. Lai, “Twitter switches from MySQL to ‘NoSQL’ database,” *Online*, 2010. .

[7] A. Lakshman, “Cassandra – A structured storage system on a P2P Network.,” *Online*, 2008. [Online]. Available: https://www.facebook.com/note.php?note_id=24413138919.

[8] A. Lakshman, “Cassandra presentation at NoSQL,” *Online*, 2008. [Online]. Available: http://www.slideshare.net/Eweaver/cassandra-presentation-at-nosql.

[9] B. Practices, M. Tavis, and P. Fitzsimons, “Web Application Hosting in the AWS Cloud,” *White Paper*, no. September, pp. 1–14, 2012.

[10] 10gen Inc, “Production Deployments,” *Online*, 2013. [Online]. Available: http://www.mongodb.org/about/production-deployments/.

[11] H. Heymann, “MongoDB at foursquare,” *Presentation*, 2010. .

[12] M. Stonebraker, “SQL databases v. NoSQL databases,” *Communications of the ACM*, vol. 53, no. 4, p. 10, Apr. 2010.

[13] M. Stonebraker and J. Hong, “Saying good-bye to DBMSs, designing effective interfaces,” *Communications of the ACM*, vol. 52, no. 9, p. 12, Sep. 2009.

[14] C. Richardson, “Orm in dynamic languages,” *Communications of the ACM*, no. June, pp. 28–37, 2009.

[15] C. Creative, “Binary JSON,” *Online*, 2012. [Online]. Available: http://bsonspec.org.

[16] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide.* O’Reilly Media, 2010.

[17] 10gen Inc, “The MongoDB 2.2 Manual,” *Online*, 2012. [Online]. Available: http://docs.mongodb.org/v2.2/. [Accessed: 03-Feb-2012].

[18] P. Eelco, M. Peter, and T. Hawkins, *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*, vol. 44, no. 5. Apress, 2010, p. 328.

[19] 10gen Inc, “An afternoon with the CEO of 10gen,” 2013. [Online]. Available: http://technoturd.wordpress.com/2013/01/14/afternoon-with-ceo-of-10gen-mongodb-company/.

[20] T. Brock, “Sharding,” *Presentation*, 2013. [Online]. Available: http://www.10gen.com/presentations/sharding-1.

[21] A. Lenk, M. Menzel, J. Lipsky, S. Tai, and P. Offermann, “What Are You Paying For? Performance Benchmarking for Infrastructure-as-a-Service Offerings,” *2011 IEEE 4th International Conference on Cloud Computing*, pp. 484–491, Jul. 2011.

[22] H. Hannon, “High Performance, Scalable MongoDB in a Bare Metal Cloud.,” *Presentation*, 2013. [Online]. Available: http://www.10gen.com/presentations/high-performance-scalable-mongodb-bare-metal-cloud.

[23] Amazon, “Amazon EC2 FAQs,” *Online*, 2012. [Online]. Available: http://aws.amazon.com/ec2/faqs/.

[24] E. Walker, “Benchmarking Amazon EC2 for high-performance scientific computing,” *Login*, vol. 33, no. 5, pp. 18–23, 2008.

[25] M. Ward, “MongoDB NoSQL Database on AWS,” *White Paper*, no. March, pp. 1–21, 2013.

[26] J. Baron, A. W. Services, and R. Schneider, “Storage Options in the AWS Cloud,” *White Paper*, no. December, pp. 1–23, 2010.

[27] A. Web and S. Risk, “Amazon Web Services : Risk and Compliance Risk and Compliance Overview,” *White Paper*, no. November, pp. 1–41, 2012.

[28] R. Schneider, “Storage Options in the AWS Cloud : Use Cases,” *White Paper*, no. December, pp. 1–12, 2010.

[29] A. W. Services, H. Aws, and P. Works, “How AWS Pricing Works,” *White Paper*, no. March, pp. 1–26, 2012.

[30] Linode, “Linode Documentation Wiki,” *Online*, 2010. [Online]. Available: http://www.linode.com/wiki/index.php/Main_Page.

[31] OpenStack Foundation, “OpenStack Operations Guide,” *Online*, 2012. .

[32] Rackspace, “Explore Cloud Servers powered by OpenStack and our next generation API.,” *Online*, 2013. .

[33] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey, “Early observations on the performance of Windows Azure,” *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC ’10*, p. 367, 2010.

[34] Windows Azure, “Install MongoDB on a virtual machine running CentOS Linux in Windows Azure,” *Online*, 2013. [Online]. Available: http://www.windowsazure.com/en-us/manage/linux/common-tasks/mongodb-on-a-linux-vm/.

[35] C. V. Hansen and J. E. Forcier, “Fabric Documentation,” *Online*, 2012. [Online]. Available: http://docs.fabfile.org/en/1.6/.

Appendix A – Fabric Code

```
import os
import json
from fabric.api import *
from fabric.contrib.files import append, contains, exists

@task
def push_ssh_key(key_file='~/.ssh/id_rsa.pub'):
    with open(os.path.expanduser(key_file)) as f:
        key_text = f.read()

    if not exists('$HOME/.ssh'):
        run('mkdir -p ~/.ssh')

    if contains('$HOME/.ssh/authorized_keys', key_text):
        print('SSH key already exists, skipping push.')
        return

    if contains('/etc/ssh/sshd_config', key_text):
        print('SSH key already exists, skipping push.')
        return

    append('~/.ssh/authorized_keys', key_text)
    run('chmod 600 ~/.ssh/authorized_keys')
    run('restorecon -R -v ~/.ssh')

@task
def install_mongodb():
    if not exists('/etc/yum.repos.d/10gen.repo'):
        put('config/10gen.repo', '/etc/yum.repos.d/10gen.repo', True)
    sudo('yum install -y mongo-10gen-2.2.3 mongo-10gen-server-2.2.3 python-
setuptools')
    sudo('chkconfig mongod on')
    sudo('service mongod start')
    sudo('easy_install pymongo')

@task
def start_mongodb():
    sudo('service mongod start')

@task
def restart_mongodb():
    sudo('service mongod restart')
```

```
def mongodb_write_perf_local(processes = 1, writeConcern = 'no-ack'):
    local('python multi-process.py --total-processes %(processes)s -
w %(writeConcern)s >> output.log' %locals())

def mongodb_read_perf_local(processes = 1):
    local('python multi-process.py --total-processes %(processes)s --read-
only-test >> output.log' %locals())

def mongodb_write_perf_remote(processes = 1, writeConcern = 'no-
ack', outputFileName = 'output-remote.log'):
    run('python /tmp/multi-process.py --total-processes %(processes)s -
w %(writeConcern)s >> %(outputFileName)s'%locals())

def mongodb_read_perf_remote(processes = 1, outputFileName = 'output-
remote.log'):
    run('python /tmp/multi-process.py --total-processes %(processes)s --read-
only-test >> %(outputFileName)s'%locals())

@task
def copy_required_scripts():
    put('multi-process.py', '/tmp/multi-process.py')
    put('multi.sh', '/tmp/multi.sh')

def mongodb_perf_remote_test(writeConcern = 'no-
ack', outputFileName = 'output-remote.log'):

    mongodb_write_perf_remote(  1, writeConcern, outputFileName)
    mongodb_write_perf_remote(  2, writeConcern, outputFileName)
    mongodb_write_perf_remote(  4, writeConcern, outputFileName)
    mongodb_write_perf_remote(  8, writeConcern, outputFileName)
    mongodb_write_perf_remote( 16, writeConcern, outputFileName)
    mongodb_write_perf_remote( 32, writeConcern, outputFileName)
    mongodb_write_perf_remote( 64, writeConcern, outputFileName)
    mongodb_write_perf_remote(128, writeConcern, outputFileName)

@task
def mongodb_perf_all_remote_tests():
    copy_required_scripts()

    outputFileName = '/tmp/output.log'

    if exists(outputFileName):
        run('rm %s' % outputFileName)

    run('nohup sh /tmp/multi.sh')

@task
def mongodb_perf_all_local_tests():
```

```
if os.path.exists('output.log'):
    local('rm output.log')

# write tests
mongodb_write_perf_local(1, 'no-ack')
mongodb_write_perf_local(2, 'no-ack')
mongodb_write_perf_local(4, 'no-ack')
mongodb_write_perf_local(8, 'no-ack')
mongodb_write_perf_local(16, 'no-ack')
mongodb_write_perf_local(32, 'no-ack')
mongodb_write_perf_local(64, 'no-ack')
mongodb_write_perf_local(128, 'no-ack')

mongodb_write_perf_local(1, 'ack')
mongodb_write_perf_local(2, 'ack')
mongodb_write_perf_local(4, 'ack')
mongodb_write_perf_local(8, 'ack')
mongodb_write_perf_local(16, 'ack')
mongodb_write_perf_local(32, 'ack')
mongodb_write_perf_local(64, 'ack')
mongodb_write_perf_local(128, 'ack')

mongodb_write_perf_local(1, 'ack-with-journal')
mongodb_write_perf_local(2, 'ack-with-journal')
mongodb_write_perf_local(4, 'ack-with-journal')
mongodb_write_perf_local(8, 'ack-with-journal')
mongodb_write_perf_local(16, 'ack-with-journal')
mongodb_write_perf_local(32, 'ack-with-journal')
mongodb_write_perf_local(64, 'ack-with-journal')
mongodb_write_perf_local(128, 'ack-with-journal')

# read tests
mongodb_read_perf_local(1)
mongodb_read_perf_local(2)
mongodb_read_perf_local(4)
mongodb_read_perf_local(8)
mongodb_read_perf_local(16)
mongodb_read_perf_local(32)
mongodb_read_perf_local(64)
mongodb_read_perf_local(128)
```

Appendix B – MongoDB Benchmark Code

```
import time
import datetime
import multiprocessing
import sys
from optparse import OptionParser
from pymongo import MongoClient
from random import shuffle

class ReadProcess(multiprocessing.Process):
    def __init__(self, lock, counter, dbName, readsArray, hostname):
        super(ReadProcess, self).__init__()
        self.kill_received = False
        self.counter = counter
        self.lock = lock
        self.dbName = dbName
        self.readsArray = readsArray
        self.connection = MongoClient(hostname)

    def run(self):
        while not self.kill_received:

            # critical section read from the array or exit
            with self.lock:
                if self.counter.value >= len(self.readsArray):
                    break
                item = self.counter.value
                self.counter.value = self.counter.value + 1

            id = self.readsArray[item] + 1

            # get the data
            document = self.connection[self.dbName].posts.find_one({'_id': id
})

            # make sure the document exists
            if document is None:
                sys.stdout.write("Document was not found.\n")
                break

class InsertProcess(multiprocessing.Process):
    def __init__(self, lock, totalInserts, totalTimeForInserts, dbName, write
Concern, total, hostname):
        super(InsertProcess, self).__init__()
        self.totalInserts = totalInserts
```

```
        self.totalTimeForInserts = totalTimeForInserts
        self.kill_received = False
        self.lock = lock
        self.dbName = dbName
        self.total = total
        self.connection = MongoClient(hostname)
        self.connection.write_concern = writeConcern

    def run(self):
        while not self.kill_received:

            # critical section update total inserts or exit
            with self.lock:
                if self.totalInserts.value + 1 > self.total:
                    break
                self.totalInserts.value = self.totalInserts.value + 1
                id = self.totalInserts.value

            # insert the data
            self.connection[self.dbName].posts.insert(self.getPostData(id))

    def getPostData(self, id):
        post = {
            "_id": id,
            "author": "Mike",
            "text": "My First Blog Post",
            "tags": ['mongodb', 'python', 'pymongo'],
            "data": datetime.datetime.utcnow()
        }
        return post

class TimerProcess(multiprocessing.Process):
    def __init__(self, totalInserts, sleepTime = 1):
        super(TimerProcess, self).__init__()
        self.totalInserts = totalInserts
        self.sleepTime = sleepTime
        self.kill_received = False

    def run(self):
        time.sleep(self.sleepTime)
        while not self.kill_received:
            sys.stdout.write("Progress: %d
records\n" % tuple([self.totalInserts.value]))
            time.sleep(self.sleepTime)

if __name__ == "__main__":

    parser = OptionParser()
```

```
    parser.add_option("-m", "--
hostname", dest="hostname", default="localhost", help="mongodb hostname
(Default localhost)")
    parser.add_option("-p", "--
port", dest="port", default=27017, help="mongodb hostname (Default 27017)")
    parser.add_option("-d", "--
db", dest="dbName", default="test_database", help="mongodb database name to
use")
    parser.add_option("-t", "--total-
processes", dest="processes", default=1, help="total number of processes to
insert with")
    parser.add_option("-n", "--num-
documents", dest="total", default=100000, help="total number of documents to
read or write")
    parser.add_option("-v", "--
verbose", action="store_true", dest="verbose", default=False, help="verbose:
true or false")
    parser.add_option("-w", "--write-
concern", dest="writeConcern", default="no-ack", help="write concern: 0, 1,
2...")
    parser.add_option("-r", "--read-only-
test", action="store_true", dest="readOnly", default=False, help="read
only?")

    (options, args) = parser.parse_args()

    # make sure options are set to the correct type
    options.hostname = str(options.hostname)
    options.port = int(options.port)
    options.dbName = str(options.dbName)
    options.processes = int(options.processes)
    options.total = int(options.total)

    # set up the write concern
    writeConcern = {}

    # set write concern
    if options.writeConcern == 'no-ack':
        writeConcern['w'] = 0
    elif options.writeConcern == 'ack':
        writeConcern['w'] = 1
    elif options.writeConcern == 'ack-with-journal':
        writeConcern['w'] = 1
        writeConcern['j'] = True
    elif options.writeConcern == 'ack-with-fsync':
        writeConcern['w'] = 1
        writeConcern['fsync'] = True
    else:
```

```
        parser.error("invalid write concern")

    # set up a connection to mongodb and remove all the results if it's a
write
    if not options.readOnly:
        connection = MongoClient(host=options.hostname, port=options.port)
        connection[options.dbName].posts.remove()

    # set up the counter value
    totalInserts = multiprocessing.Value('i', 0)
    totalTimeForInserts = multiprocessing.Value('d', 0)

    lock = multiprocessing.Lock()

    # set up the array for reads
    x = range(options.total)
    shuffle(x)
    readsArray = multiprocessing.Array('i', x)

    # set up the processes but don't start them
    processes = []
    for i in range(options.processes):
        if options.readOnly:
            p = ReadProcess(lock, totalInserts, options.dbName, readsArray, o
ptions.hostname)
        else:
            p = InsertProcess(lock, totalInserts, totalTimeForInserts, option
s.dbName, writeConcern,options.total, options.hostname)
        p.daemon = True
        processes.append(p)

    # set up the timer thread if verbose is enabled
    if options.verbose:
        tProcess = TimerProcess(totalInserts)
        tProcess.daemon = True
        tProcess.start()

    # start the processes
    start = time.time()
    for p in processes:
        p.start()

    try:
        # wait for all the processes to finish or be terminated
        while len(processes) > 0:
            for process in processes:
                process.join(3600)
```

```
            processes[:] = [process for process in processes if process.is_al
ive()]

    except KeyboardInterrupt:
        sys.stdout.write("Bye\n")

    end = time.time()

    # kill the timer process
    if options.verbose:
        tProcess.kill_received = True

    stats = {
        'writeConcern': options.writeConcern,
        'num': totalInserts.value,
        'time': end - start,
        'processes': options.processes,
        'rps': totalInserts.value / (end - start) if end - start > 0 else 0
    }
    if options.verbose:
         sys.stdout.write("---------------\n")

    if options.readOnly:
        sys.stdout.write("read %(num)d records in %(time).2f seconds
using %(processes)d processes. (avg %(rps).2f documents per second)\n" %
stats)
    else:
        sys.stdout.write("%(writeConcern)s: wrote %(num)d records
in %(time).2f seconds using %(processes)d processes. (avg %(rps).2f documents
per second)\n" % stats)
```