MongoDB NoSQL Database on AWS

Miles Ward

March 2013

Introduction

Amazon Web Services (AWS) is a flexible, cost-effective, easy-to-use cloud computing platform. NoSQL software packages are widely deployed in the AWS cloud. Running your own NoSQL data store on Amazon Elastic Cloud Compute (Amazon EC2) is a great scenario for users whose application requires the unique benefits of structured storage software optimized for high-performance operations on large datasets.

This white paper will help you understand one of the most popular NoSQL options available with the AWS cloud computing platform, the open source application MongoDB. We provide an overview of general best practices that apply to all major NoSQL options, and we examine important MongoDB implementation characteristics such as performance, durability, and security. We pay particular attention to identifying features that support scalability, high-availability, and fault-tolerance.

What is NoSQL?

NoSQL is a popular name for a subset of structured storage software that is designed is optimized for high-performance operations on large datasets. This optimization comes at the expense of strict ACID (atomicity, consistency, isolation, and durability) compliance and, as the name implies, native querying in the SQL syntax. NoSQL software is easy for developers to use, horizontally scalable, and optimized for narrow workload definitions.

There are three major categories of NoSQL applications today:

- Key-value stores like Cassandra, Riak, and Project Voldemort
- Graph databases like Neo4j, DEX, and InfiniteGraph
- Document stores like MongoDB, CouchBase, and BaseX

These NoSQL applications are either separate open source software (OSS) projects or commercial closed-source projects. The applications are written in different languages, they expose different interfaces, and they implement different optimizations.

NoSQL on Amazon EC2

Amazon Web Services (AWS) provides an excellent platform for running many advanced data systems in the cloud. Some of the unique characteristics of the public cloud provide strong benefits for NoSQL workloads. In many ways, AWS infrastructure is similar to a local, physical infrastructure, but there are some differences. A general understanding of those differences can assist greatly in making good architecture decisions for your system.

Amazon Web Services has native NoSQL services that do not require direct administration and offer usage-based pricing. Consider these options as possible alternatives to building your own system with an OSS or a commercial NoSQL application.

Amazon Simple Storage Service (Amazon S3) provides a simple web services interface that can store and retrieve any amount of data at any time from anywhere on the web. Amazon S3 gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of websites. Amazon S3 maximizes benefits of scale and passes those benefits on to you.

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. All data items are stored on solid-state drives (SSDs) and are automatically replicated across three Availability Zones in an AWS region to provide built-in high availability and data durability. With Amazon DynamoDB, you can offload the administrative burden of operating and scaling a highly available distributed database cluster while paying a low variable price for only the resources you consume.

Performance

The performance of a NoSQL system on Amazon Elastic Compute Cloud (Amazon EC2) depends on many factors, including the Amazon EC2 instance type, the number and configuration of Amazon Elastic Block Store (Amazon EBS) volumes, the configuration of the NoSQL software, and the application workload. We encourage you to benchmark your actual application on several Amazon EC2 instance types and storage configurations in order to select the most appropriate configuration.

To increase the performance of your system, you need to know which of the server's resources is the performance constraint. If CPU or memory limits your system performance, you can scale up the memory, compute, and network resources available to the software by choosing a larger Amazon EC2 instance type. Remember, 32-bit Amazon Machine Images (or AMIs) can't run on 64-bit instances, so if you anticipate needing the higher-performance instance types, start with a 64-bit instance. Remember, too, that changing an existing instance to a different size requires a stop/start cycle.

If your performance is limited by disk I/O, you might consider changing the configuration of your disk resources. Amazon EBS volumes, the persistent block storage available to Amazon EC2 instances, are connected over the network. An increase in network performance can have a significant impact on aggregate performance, so be sure to choose the appropriate instance size. If your instance supports the EBS-Optimized flag, enabling it will add 500 Mbps or 1000 Mpbs (depending on instance type) of bandwidth to EBS. To scale up random I/O performance beyond the roughly 100 IOPS (input output operations per second) that a single standard EBS volume can deliver, you can leverage Provisioned IOPS volumes, which allow configuration up to 2000 IOPS per volume.

If you require even greater concentrations of random I/O, you can increase the number of EBS volumes as a ratio of EBS storage (8 x 250 GB EBS volumes versus 2 x 1000 GB EBS volumes). If aggregation is required, you can use software RAID (redundant array of independent disks) 0 (disk striping) or RAID 10 (a mirror of stripes) across multiple EBS volumes to increase single logical volume total IOPS. Remember, utilizing RAID 0 striping reduces operational durability of the logical volume by a degree inversely proportional to the number of EBS volumes in the stripe set. Similarly, RAID 10 provides increased redundancy and, in the event of a volume failure, the ability to replace a single EBS volume without application downtime. RAID 10 also offers increased read throughput for each stripe of data, at the expense of a 50% reduction of the provisioned aggregate write performance. Evaluate very carefully the redundancy model you select for your applications.

A single standard EBS volume can provide approximately 100 IOPS steady state, and single instances with arrays of 10 or more attached EBS disks can often reach 1000 IOPS sustained. Provisioned IOPS EBS volumes allow up to double this performance with a single provisioned volume without the increased failure risk associated with RAID 0 aggregation. With four Provisioned IOPS EBS volumes configured for 2000 IOPS on m2.4xlarge or m1.xlarge EBS-Optimized instances, 8000 read and write IOPS are available on a single instance, which would roughly saturate the EBS-Optimized network link to EBS. In order to achieve this performance, block sizes must be limited to 16 KB, and queue depths of greater than 5 requests per 1000 IOPS must be maintained. Because Provisioned IOPS can process I/O requests smaller than this 16 KB limit, applications with an average read/write block size smaller than 16 KB can improve performance by further increasing the number of Provisioned IOPS volumes. For example, on the same hosts described above, if your block size average is 8 KB, eight Provisioned IOPS volumes could be attached to deliver up to 16,000 8 KB IOPS.

For sequential disk access, ephemeral disks offer somewhat higher performance and don't impact your network connectivity. Some customers use ephemeral disks in conjunction with Amazon Elastic Block Store to conserve network bandwidth and EBS I/O for more random, small-block operations.

In many cases, the constraint is more abstract. In general, you can use the same system performance tuning options in the Amazon EC2 environment that you use in a physical server environment. You can also scale the total performance of a NoSQL system by scaling horizontally across multiple servers with sharding, caching, and synchronous and asynchronous replication strategies.

Durability and Availability

No approach to structured storage is complete without using the available options in each system for application-level redundancy. The Amazon EC2 environment and Amazon NoSQL services provide the tools you need to deliver highly durable services, as the following table illustrates:

AWS Service	Durability Design
Amazon S3 backups	Designed to provide 99.999999999% durability and 99.99% availability of objects over a given year
Amazon EBS volumes	Amazon EBS volumes with 20 GB or less of modified data since their most recent Amazon EBS snapshot can expect an annual failure rate (AFR) of between 0.1% – 0.5%, where failure refers to a complete loss of the volume. For more information, see "Amazon EBS Volume Durability" on the Amazon Elastic Block Store home page.
Amazon EC2 instances	Can detach volumes and reattach to new instances of Amazon EC2 servers upon instance failure; rapid access to replacement instances
Regions	Can deliver individual zone independence by using multiple Availability Zones
Amazon Web Services	Redundant and fault-tolerant API control layer. Amazon EC2 API has an excellent SLA.

Elasticity and Scalability

In many cases, users of NoSQL solutions on Amazon EC2 can take advantage of the elasticity and scalability of the underlying AWS infrastructure. For example, once you have configured an Amazon EC2 instance with your structured storage application, you can bundle the instance into a custom AMI and then create multiple new instances of your configuration within a few moments.

For many customers, selecting an instance class with higher performance is the easiest way to increase application performance overall. Selecting higher and higher performance single instances is often referred to as *scaling up*. Scaling up is particularly useful if you have a set maintenance window and can tolerate system downtime. More advanced scaling, sharding, or otherwise spreading the database query load across multiple Amazon EC2 instances, can provide even higher performance. Increasing performance by deploying additional instances and sharing the load across them is often referred to as *scaling out*.

Configuration

To create a NoSQL structured data store on an Amazon EC2 instance, you can launch an instance from an AMI that has the base operating system you want and then install the application software using the standard installation software provided by the operating system. Remember, there's no DVD drive in the cloud, so you have to download the required software.

After your application is installed and configured on an Amazon EC2 instance, you can interact with your instance by using its management interface, either by exposing that interface over the web (or a VPN) to your location or by remotely accessing the server over SSH, NX, VNC, or RDP. You can work with an application on Amazon EC2 just as you would with an on-premises software application. You'll need to configure the instance's security group to allow traffic on the port that is used by the NoSQL application. All ports are disabled by default, so make sure you open up the ports you need.

The Amazon EBS volumes that serve as the root drives of Amazon EC2 instances are natively redundant, and they are designed to survive the physical failure of the Amazon EC2 instance or an individual EBS hardware failure; however, you might have complex operating system and application configurations that should be preserved by rebundling your AMI. When you rebundle your AMI, subsequent or additional launches of Amazon EC2 instances will include all of your configuration changes. For directions, go to Creating Amazon EBS-backed AMIs in the *Amazon Elastic Cloud Compute User Guide.*

Pricing

Not having to manage the hardware and having the ability to quickly provision capacity to scale your NoSQL structured storage makes the AWS cloud the optimum solution for running a NoSQL system. The AWS cloud not only gives you design flexibility, which we'll cover in greater detail in coming sections of this paper, but it also reduces operational costs. Depending on the architecture of your NoSQL solution, there are a few factors you should account for when estimating the cost of delivery:

1. Compute: Amazon EC2 instance-hours purchased either in the on-demand, Reserved (Light, Medium or Heavy Utilization), or Spot models for the computational load of the system.

2. Storage: Amazon EBS (both standard and Provisioned IOPS) volumes for your storage (both actual storage capacity and I/O to disk).

3. Data Transfer: Network transit, particularly inter-Availability Zone networking charges for configurations that span multiple availability zones, such as replica sets).

4. Backups: Amazon S3 storage for snapshots of configured root and data (EBS) volumes.

Feel free to leverage tools like the AWS Simple Monthly Calculator to model your projected workloads.

MongoDB on Amazon EC2

Overview

MongoDB is a scalable, high-performance, open source, structured storage system. MongoDB provides JSON-style document-oriented storage with full index support, sharding, sophisticated replication, and compatibility with the Map/Reduce paradigm. MongoDB focuses on flexibility, power, speed, and ease of use.

> **Note** The procedures in this white paper use two different prompts. Items that begin with $ are run at a standard Linux shell prompt, which may require syntax adjustment on Windows systems. Items that begin with > are run from the `mongo` shell and represent commands to the `mongod` process.

Basic tips

- MongoDB is updated frequently; www.mongodb.org is a critical resource for the latest code enhancements, documentation, and news on this open source software (OSS) project.

- 10gen develops MongoDB and offers subscriptions, which include enterprise features that are not in the open source code, production support, training, and consulting for MongoDB.

- Use 64-bit instances only. MongoDB can store only about 2.5 GB of data in a 32-bit system. For more information, go to http://blog.mongodb.org/post/137788967/32-bit-limitations.

- T1.micro instances are not recommended for production MongoDB deployments, including arbiters, config servers, and mongos shard managers.

- Use XFS or Ext4. These file systems support I/O suspend and write-cache flushing, which is critical for multidisk consistent snapshots, as well as important tuning options to improve MongoDB performance.

- Use Provisioned IOPS EBS volumes in RAID 10. MongoDB is specifically optimized around the benefits of RAID 10, and for all but the most write-heavy workloads it is the most efficient configuration.

- Be careful with large objects. MongoDB currently restricts documents to 16 MB. If you need to accommodate larger objects, use GridFS within Mongo. For more information, go to http://www.mongodb.org/display/DOCS/GridFS.

- Raise file descriptor limits. The default of 1024 on most systems won't work for production-scale workloads. For more information, go to http://www.mongodb.org/display/DOCS/Too+Many+Open+Files.

- Use MongoDB 2.2 or later. Significant feature enhancements in the 2.0 release include security in sharded environments and other critical elements.

- Turn off atime and diratime when you mount the data volume. Doing so reduces I/O overhead by disabling features that aren't useful to MongoDB.

- Don't use large virtual memory (VM) pages. For more information see, http://linuxgazette.net/155/krishnakumar.html.

Basic MongoDB Installation

1. Launch an Amazon EC2 instance using the AMI of your choice. (Our example uses the Amazon Linux 64-bit AMI.)
2. Create an Amazon EBS volume to use for your MongoDB storage, and attach it to the instance.
3. Connect to the instance via SSH.
4. Make a file system on your EBS volume.

   ```
   $ sudo mkfs -t ext4 /dev/*the_connection_you_attached_the_volume_to_for_example_sdf*
   ```

5. Make a directory to which the file system can be mounted.

   ```
   $ sudo mkdir -p /data/db/
   $ sudo chown `id -u` /data/db
   ```

6. Edit your `fstab` to enumerate the volume on startup.

   ```
   $ sudo echo '/dev/sdf /data/db auto noatime,noexec,nodiratime 0 0' >> /etc/fstab
   ```

7. Mount the volume.

   ```
   $ sudo mount -a /dev/sdf /data/db -o noatime,noexec,nodiratime
   ```

8. Install MongoDB with a package manager: in order to do so, create at the following file:

   ```
   /etc/yum.repos.d/10gen.repo
   ```

9. Edit the file to contain the following:

   ```
   [10gen]
   name=10gen Repository
   baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64
   gpgcheck=0
   ```

10. After you've created the file, run the following command:

    ```
    $ sudo yum install mongo-10gen mongo-10gen-server
    ```

11. Edit the /etc/mongodb.conf file to increase oplogSize, enable journaling, and target your data and log directories.

    ```
    dbpath = /data/db
    logpath = /data/db/logs
    ```

12. Use the AWS Management Console to allow ingress from your application servers over appropriate ports by creating rules in your Amazon EC2 security group. (The default port is 27017.) For information about the AWS Management Console, see <link>.
13. Start `mongod`.

    ```
    $ service start mongod
    ```

Architecture

The design of your MongoDB installation on Amazon EC2 depends on the scale at which you want to operate. Are you experimenting with the framework on your own for a private project? If so, you'll probably install `mongod` (the running MongoDB application) on the same machine as the rest of your software and collocate the `mongod` file structure with the rest of your application on the root volume of the instance. This scale is fine for developer experimentation; however, for production deployments, replica sets and in some cases, sharding, should be used to provide higher performance and fault tolerance.

MongoDB has two separate constructs for multinode topologies. These constructs, replica sets and sharded replica sets, are often combined in the highest-performance systems. Replica sets are an asynchronous cluster replication and automated failover technology, and sharding is an automatic data distribution system. Increasing the number of instances in a replica set provides fault tolerance. By increasing the number of shards (each one being a replica set), you can distribute distinct data to separate `mongod` processes, which provides horizontal scalability for read and write performance.

The `mongos` process routes each request to the appropriate shard, which provides seamless access to your distributed dataset. You can use the `ReadPreference` attribute in your requests if you want to enable routing of requests to secondary members.

Building Blocks

mongod

`Mongod` is the primary MongoDB application. It is responsible for the structured storage system. `Mongod` will be running on each replica of the cluster that you deploy.

mongos

`Mongos` is a routing and coordination process that makes the `mongod` nodes in the cluster look like a single system. `Mongos` processes don't store data; rather, they pull their state from the `configserver` process on startup. Any changes that occur on the `configserver` processes are propagated to each `mongos` process. `Mongos` processes do store some state, for example, open cursors from clients. Losing a `mongos` process will invalidate these cursors, but it will not result in any data loss.

`Mongos` processes can be run on the shard servers themselves, but they are lightweight enough to exist on each application server. Many `mongos` processes can be run simultaneously, because these processes do not coordinate with one another.

configserver

`Configserver` is a `mongod` process that synchronously replicates the state information of a sharded environment. Although `configserver` can run stand-alone, for data safety and high availability, any production deployment should have three `configserver` processes that have copies of the same metadata. The `configserver mongod` process is fairly lightweight, but it does require CPU and disk resources. Sizing a `mongos` server will depend on the volume of data, number of shards, number of collections, and number of `mongos` servers in your deployment. The data in the `configserver` process is critical to the cluster operation, and care should be taken not to lose all three of your `configserver` processes or their data.

To change the IP address of `configserver` processes, you may need to take the cluster down or change the location to which the host name points. An easier approach is to have DNS records for each `configserver` instance so that you can change the locations easily. This approach introduces a dependency on your DNS solution. Amazon Route

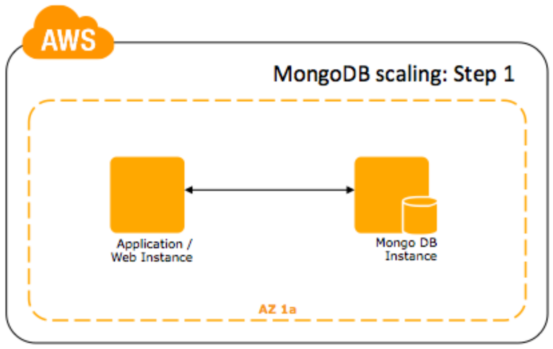
53 provides a highly available, high-performance DNS service inside Amazon EC2 that's very easy to set up, or you can run a BIND or another DNS server on our network.

The `configserver` is only required in sharded deployments. If you are just running a replica set, there is no need for `configserver`.

Production Designs

So, given these building blocks, how would a system scale to accommodate growing load over time? The following steps and diagrams guide you through a sample production design process that scales as the load grows.

Step 1: Separate the `mongod` server from the application tier. In this setup, there's no need for `configserver`, or `mongos`, because there is no sharding structure for your applications to navigate.

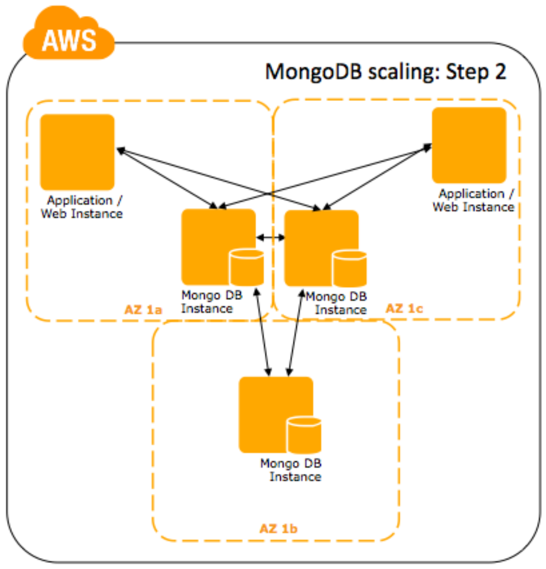


Step 1: provision a separate instance for MongoDB

Step 2: For replica sets, run multiple `mongod` instances across separate availability zones (or regions).

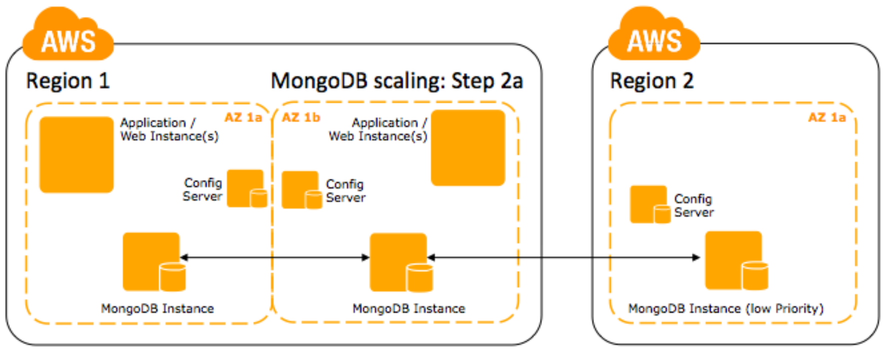
> **Note**: MongoDB replication is asynchronous; if you want to ensure that writes are propagated to multiple nodes before the primary node accepts them, use a Write Concern. For more information, see Write Concern in the MongoDB documentation.

Use an odd number of MongoDB instances. In this example we use three, but you can use more if you need higher read performance. When MongoDB detects that the primary node in the cluster has failed, it performs an election to determine which node will be the new primary. It is critical that a majority can be reached in that election. In order to prevent the election failure, an odd number of nodes should be chosen. To ensure high availability, these replicas should be placed in different Availability Zones.



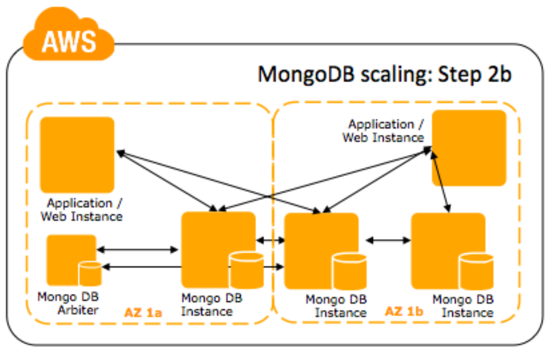
Step 2: Replicate MongoDB across Availability Zones

Step 2a: If your desired region supports only two Availability Zones, consider replication to the next nearest region, configuration of that replica with a low priority, and WAN optimization of the cross-regional connection.



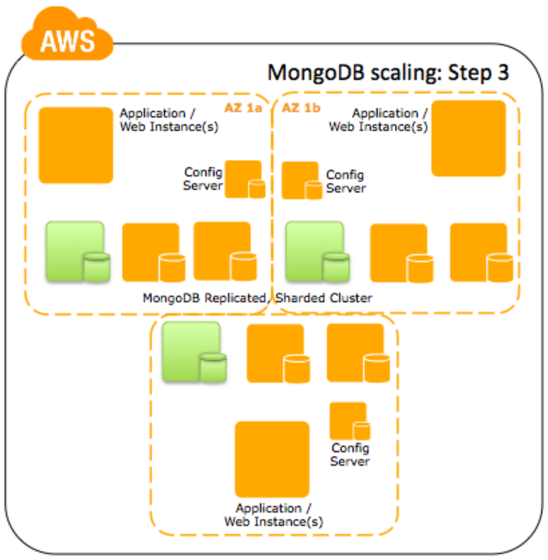
Step 2a: Leverage Remote regions in Regions where there are less than 3 AZ's available

Step 2b: Alternatively, you could place an arbiter instance in the Availability Zone with the fewest instance members of the replica set. Having an arbiter instance allows failover when a majority of replicas (2/3 in this example) are lost.



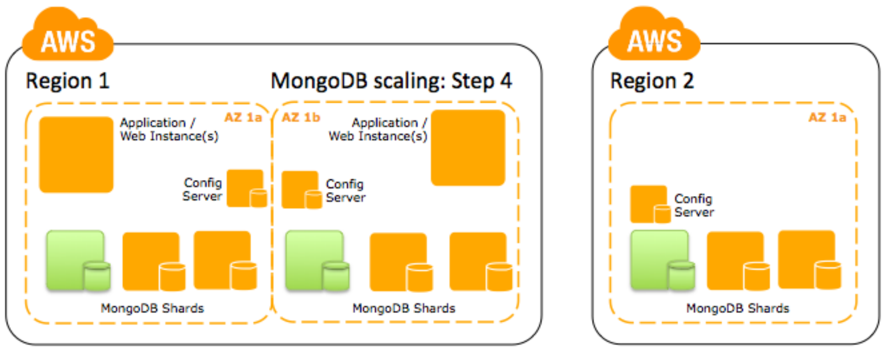
Step 2b: Leverage arbiter instances as an alternate to multi-regional replication

Step 3: Create a sharded cluster of replica sets. In this example, three `configserver` processes are running on separate m1.small or larger EC2 instances, each in separate Availability Zones.


AWS
MongoDB scaling: Step 3
AZ 1a
AZ 1b
Application / Web Instance(s)
Config Server
MongoDB Replicated, Sharded Cluster

Step 3: Increase horizontal scale by sharding MongoDB across Instances

Step 4: Scale to a multi-region system that uses a time-delayed replica for warm backup. This isn't a higher-scale step, but it is a pattern for higher availability.



Step 4: Provision a multi-regional replicated, sharded MongoDB cluster on EC2 Instances

Additional Scaling Considerations

Vertical scaling doesn't offer the same benefits as horizontal scaling. Vertically scaled higher-performance instances can provide many of the performance benefits of a more complex replicated and sharded topology, but remember that vertically scaled instances come with none of the significant fault-tolerance benefits. While it's critical for most use cases to provide the `mongod` process with sufficient memory and disk IOPS, remember that you have a virtually unlimited pool of those resources if you can scale horizontally.

Scaling step by step when you know you need a big system isn't efficient in MongoDB. There are significant time-to-value benefits to launching the full set of shards you're expecting to run or increasing the count of instances in the replica set all at once. Avoid incremental steps if you can.

Similarly, scaling under maximum load isn't a good idea. MongoDB needs significant compute, storage, and network resources in order to expand its footprint. Waiting until you're at 99 percent load to start the sharding process won't help; it simply adds more work to the system at the least opportune time. We recommend that you make changes earlier and scale out sooner, while you still have capacity to spend on the transition. One potential workaround you can try if you're caught in an overload situation is to build your new fleet of instances from backups of production instances, take a small maintenance window to sync the new more powerful fleet, and then return to service.

Designing Storage to Support MongoDB

The pattern of scaling the instance footprint for a MongoDB cluster is designed to deliver additional process memory to your workload. You'll want to design your infrastructure carefully to avoid storage bottlenecks and make the best use of your available storage resources.

MongoDB has several internal processes that require fast access to disk but with different access patterns.

The journal – The journal is a transaction log of updates to the database. Every update operation is stored in the journal before it is accepted. The journal is periodically flushed to disk. This flush typically happens every 100 ms, although it sometimes happens more often, for example, if active operations are explicitly waiting on a disk flush. The size of each journal flush will depend on the volume of data being written to MongoDB (e.g., the amount of data updated in the last 100 ms).

Background flushing – MongoDB uses `mmap` to access database data that is backed by your file system. After updates are written to the journal, they are applied in memory to the `mmap`ped version of the data files. In order to synchronize the changes in memory to the on-disk version of data, MongoDB periodically synchronizes the in-memory version of the data with the on-disk version. By default, this happens every 60 seconds, but you can tune this interval with the `syncDelay` option. The actual volume of data synced during this process will depend on your workload. If you update lots of random objects in MongoDB, you will need to write lots of random pages of memory during the sync. On the other hand, if your workload is updating a single object repeatedly, it's possible that only the single page holding that object needs to be synced.

Page faults – In order for MongoDB to process an operation on an object, that object must be resident in memory. When you perform a read or write operation on an object that is not currently in memory, a page fault is triggered. In a page fault, the operating system fetches that page from disk and loads it into memory. If your application's working set is much larger than RAM, then access requests to some objects will cause page faults before the operation can complete. Similarly, if you start MongoDB cold (e.g., where there is no data loaded into the page cache), there is a warmup period where operations can trigger a lot of page faults. (Restarting the `mongod` process does not result in a "cold start" scenario, because the pages are actually cached by the operating system and thus still in memory when the process restarts.) Page faults are often the largest driver of random I/O, especially for databases that are larger than available memory.

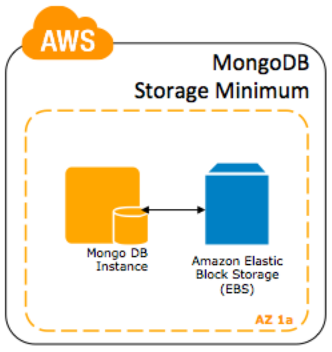
The size of I/O requests on underlying block devices is typically governed by your choice of operating system, file system, and block device settings. In particular, you should pay attention to the read-ahead settings on your block device to see how much data is read when a page fault occurs. Having a large setting for read ahead is typically ill advised, as it will cause the system to read more pages than necessary into memory and evicting other pages that may be useful to your application. This is particularly true for services that limit block size, such as Amazon Elastic Block Store Provisioned IOPS volumes.

Understanding the total I/O load created by your journal, background flush, and page fault activity is key to selecting an appropriate storage configuration. Most of the I/O driven by MongoDB is random, and so the thing you should be paying attention to is how many IOPS are required to support your workload. If your working set is much larger than memory with random access patterns, you may need many thousands of IOPS from your storage layer to satisfy demand.

Once you have determined the IOPS load of your database, Provisioned IOPS volumes provide an excellent solution to building a high-performance storage layer for your MongoDB instance. Regular EBS volumes are not designed to provide a high ratio of I/O to provisioned storage. On the other hand, Provisioned IOPS volumes support specifically configured IOPS rates, making it much easier to predict the expected performance of a system configuration. Provisioned IOPS volumes can be provisioned with up to 2000 IOPS, and software RAID can be used to tie these volumes together into arrays that exhibit even higher performance.

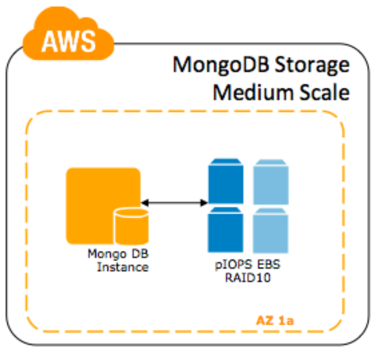
So, how should you grow? These are some efficient configurations for storage that increase instance density. Your specific workload might demand a different setup, but these are good places to start.

Minimum scale – Use Amazon Elastic Block Store (Amazon EBS). Amazon EBS has significant write cache and improved random I/O performance, and it provides greater durability than the ephemeral disks on the Amazon EC2 instance. If you're going to use ephemeral disks on instances that expose more than a single volume, be sure to mirror those volumes (using RAID 1 or RAID 10 on m1.xlarge) to enhance operational durability. Remember, even though the disks are mirrored, if the instance is stopped, fails, or is terminated, you'll lose all of your data. Using Amazon EBS volumes is a great design for systems that deliver fewer than 100 MongoDB operations per second or are tolerant of storage performance variability.



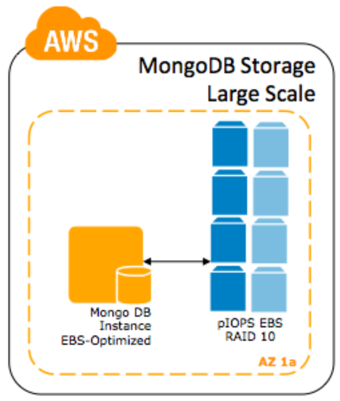
Minimum Scale: Use EBS

Medium scale – To optimize for high durability and read scaling, use RAID 10 on EBS Provisioned IOPS volumes. For best volume performance at this scale, use an m1.large instance with the EBS-Optimized flag set. We suggest using four volumes for the `mdadm` RAID 10 and creating LVM volumes within that `mdadm` volume for your /data/db file, journal, and log files.



Medium Scale: Use pIOPS RAID 10

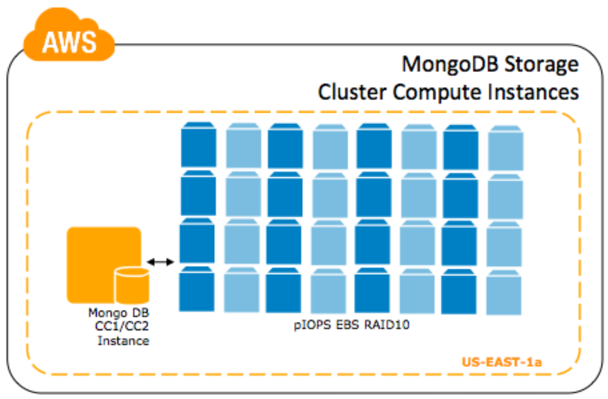
Big scale – Move up to higher-bandwidth EBS-Optimized instance types (m1.xlarge, m2.4xlarge provide 1000Mbps to EBS), and increase the size of the RAID 10 to be an 8-volume set. At this level of performance, you'll notice that you are just slightly throughput-limited to Amazon EBS by Amazon EC2 networking during atypical sequential workloads.



Big Scale: Use 8-volume pIOPS RAID 10

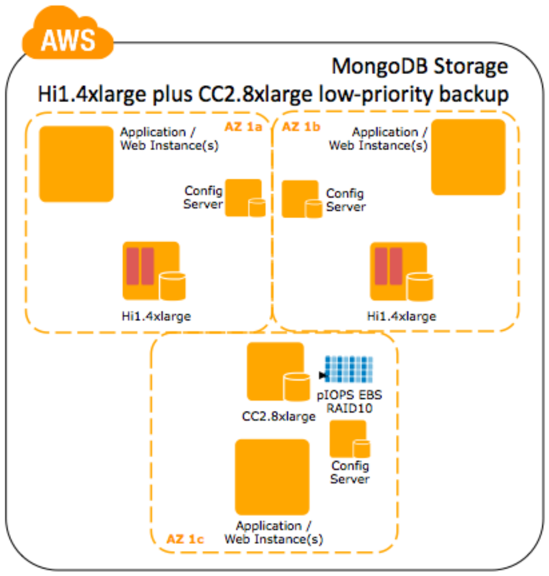
Extra-Big scale – In the very highest-density systems, you will probably increase your optimization around the specifics of your workload. In doing so, you'll take advantage of Amazon EC2 options that have some tradeoffs. Here are three different standard approaches:

- **CC2.8xlarge** – For workloads in the us-east-1, us-west-2, and eu-west-1 regions, we can provide instances with slightly less memory than our m2.4xlarge instances but with even higher network bandwidth to Amazon EBS volumes and much higher node-to-node bandwidth for replication. Assuming optimization around the typical 4 KB workload, these instances can exploit 24 or more Provisioned IOPS volumes at 2000 IOPS apiece, rivaling the performance of our hi1.4xlarge instances with higher redundancy.



Extra-BIg Scale: Use CC-class Instances, and 24+ volume RAID 10

- **Hi1.4xlarge** – For workloads with a high bias to reads and very large datasets, high I/O instances can deliver more than 120,000 4 KB random read IOPS and between 10,000 and 85,000 4 KB random write IOPS (depending on the active logical block addressing span) to applications across two 1 TB local SSD data volumes. These volumes can be configured in either RAID 1 (reducing write throughput by 50%) or RAID 0. Exercise caution when using hi1.4xlarge instances as primary storage for data; as with all ephemeral disks, stop commands or instance failures will cause irrecoverable loss of data. Snapshots are not available on ephemeral disks, so we strongly recommend operating with hi1.4xlarge replicas and adding another replica that uses EBS volumes (on a CC2 instance), with replication for greatly simplified durable backups to Amazon Simple Storage Service (Amazon S3) by snapshot.

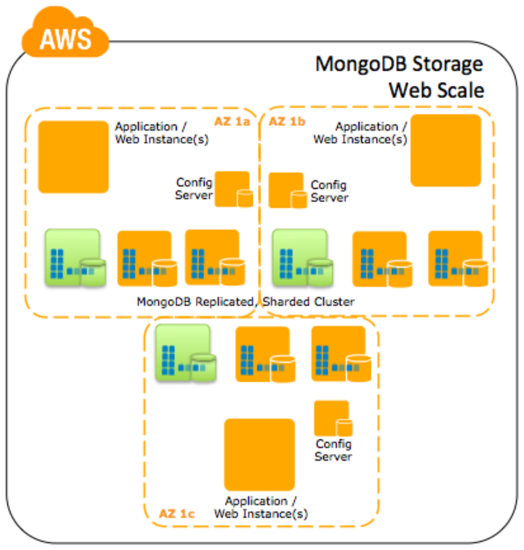

AWS
MongoDB Storage
Hi1.4xlarge plus CC2.8xlarge low-priority backup
AZ 1a
Application / Web Instance(s)
Config Server
Hi1.4xlarge
AZ 1b
Application / Web Instance(s)
Config Server
Hi1.4xlarge
CC2.8xlarge
pIOPS EBS RAID10
Config Server
Application / Web Instance(s)
AZ 1c

Hi1.4xlarge: Use Replication, and a low-priority replica with EBS

- **CR1.8xlarge** – A CR1.8xlarge instance provides the current maximum Amazon EC2 memory footprint at 244GB of usable memory, which offers a very significant performance advantage for MongoDB datasets that are larger than the amount of memory that is available in other instance types. These instances also provide a pair of local 120 GB SSDs that operate at a lower performance than the Hi1.4xlarge instance above. If your dataset is greater than 60 GB but smaller than 240 GB, using the local ephemeral volumes can provide a performance boost over an Amazon EBS Provisioned IOPS volume. If your dataset exceeds 240 GB per node, use of Provisioned IOPS volumes following the cc2.8xlarge pattern is strongly recommended.

- **M2.4xlarge** – In regions that don't offer the former instance types, increasing to 12 Provisioned IOPS volumes at 2000 IOPS each represents the upper bound for 4 KB operations on EBS-optimized hosts. At this level, you will be significantly performance limited by network during the atypical sequential workloads, which means that replication or backup restores won't happen any faster than they would in our big scale design. If your workload is strongly write-biased, you might experiment with RAID 0, remembering that this increases the likelihood of volume-group failure and will probably increase your reliance on MongoDB's replica-set technology.

Web Scale – Shard your MongoDB installation across multiple EC2 Instances. Read performance at the scale supported by MongoDB can be strongly limited if the working dataset isn't in memory, so make sure you have sufficient shard breadth to prevent excessive page faults. Choose the instance configuration from the Extra-Big options above that best match your use case, and create enough instances to meet your requirements.


AWS
MongoDB Storage
Web Scale
AZ 1a
AZ 1b
AZ 1c
Application / Web Instance(s)
Config Server
MongoDB Replicated, Sharded Cluster

Web Scale: Shard MongoDB using Extra-Big storage configurations

Creating a Replica Set

Setting up asynchronous replication in MongoDB is a three-step process:

1. Provisioning the requisite instances and storage
2. Installing and starting the `mongod` processes
3. Formally initiating the set

To create a replica set

1. Start three identical EC2 instances, each with enough disks for the medium-scale recommendation discussed earlier.

```
$ ec2-run-instances ami-1b814f72 -b /dev/sdf=:80:io1:500 -b /dev/sdg=:80:io1:500 -b
/dev/sdh=:80:io1:500 -b /dev/sdi=:80:io1:500 -g smongo -m -t m1.xlarge --ebs-optimized true
-z us-east-1a

$ ec2-run-instances ami-1b814f72 -b /dev/sdf=:80:io1:500 -b /dev/sdg=:80:io1:500 -b
/dev/sdh=:80:io1:500 -b /dev/sdi=:80:io1:500 -g smongo -m -t m1.xlarge --ebs-optimized true
-z us-east-1b

$ ec2-run-instances ami-1b814f72 -b /dev/sdf=:80:io1:500 -b /dev/sdg=:80:io1:500 -b
/dev/sdh=:80:io1:500 -b /dev/sdi=:80:io1:500 -g smongo -m -t m1.xlarge --ebs-optimized true
-z us-east-1c
```

2. Follow the basic installation instructions discussed earlier, with the following alteration to the step 14, start `mongod`:

```
$ mongod --replSet thenameofyourreplicaset --port 27017 --dbpath /db/data
```

3. When you run the mongod command, the node should output an error message saying that the replica set isn't up and running yet. You can use DNS to swap out individual replica set hosts, so create three DNS entries in Amazon Route 53 or your DNS system of choice. For more information, go to Creating, Changing, and Deleting Resource Record Sets in the *Amazon Route 53 Developer Guide*. Alternatively, you could host the MongoDB

instances inside the Amazon Virtual Private Cloud (Amazon VPC), where you have control over static instance IP addresses.

4. Start the `mongo` shell on any one of the replica hosts, create the configuration for the replica set, and then initiate the replication.

```
$ Mongo

$ > config = {_id: 'foo', members: [
                   {_id: 0, host: 'replicaset1.mongohosts.mysite.com:27017'},
                   {_id: 1, host: 'replicaset2.mongohosts.mysite.com:27017'},
                   {_id: 2, host: 'replicaset3.mongohosts.mysite.com:27017'}]}
$ > rs.initiate(config);
```

   You can also pass a `priority:value` flag for each node. This flag controls the priority by which nodes will be elected to the master in case of a node failure. Setting `priority:value` is useful in designating nodes for running backups or for spanning regions. That way, if one of the two nodes in the primary region fails, the other node in that region (not the distant disaster recovery (DR) node) would be elected as the primary node. For more information, see the MongoDB documentation at: http://www.mongodb.org/display/DOCS/Replica+Set+Tutorial and http://www.mongodb.org/display/DOCS/Data+Center+Awareness

Creating a Sharded and Replicated MongoDB Cluster

In order to support high availability of the architecture and both read and write performance scalability, sharded systems should be built on top of replicated systems.

To create a sharded MongoDB cluster

1. Start additional hosts for `mongod`, setting the `--shardsvr` flag to enable sharding, and supplying an `_id` name to distinguish each replica set. Enabling sharding changes the default port to 27018. For example, you could have three shards with three replicas (a primary and two secondaries) each, for a total of nine EC2 instances.

2. Confirm that you've started `configserver` processes by setting the `--configsvr` flag to enable the `configserver` process. Enabling the configserver process will set the default port to 27019. These config servers can be distributed among the `mongod` servers at random, or they can run on stand-alone instances (m1.small or greater, depending on total cluster size). Config servers should be spread across Availability Zones to ensure that they do not fail together. If you are running MongoDB across multiple regions, you should also distribute config servers across those regions.

3. Install `mongos` routers on each application host, starting them with the `--configdb` flag, passing the DNS names and ports for all three of the `config server` instances.

4. Enable sharding on the databasein order to enable the `shardcollection` command in the following step.

```
> db.runCommand( { enablesharding : "<dbname>" } );
```

5. Shard the relevant collections. The key is the object in the collection that will be indexed and used to distribute chunks of items across your cluster. Once a collection is sharded, it cannot be resharded without a backup/restore cycle, so take care when you select a shard key.

```
> db.runCommand(
{ shardcollection : "<namespace>",key : <shardkeypatternobject> });
```

Operations

Backup

To run MongoDB on Amazon EC2, we strongly recommend that you use MongoDB version 2.0.7 or later and that you set the --journal option. We also recommend placing the journal on Amazon EBS Provisioned IOPS volumes. The actual mechanics of backup are greatly simplified by the Amazon S3 snapshot capability native to Amazon EBS.

To perform a backup

1. Flush and lock the `mongod` by using the `fsync` and the `lock` commands.

```
> db.runCommand({fsync:1,lock:1}); {    "info" : "now locked", "ok" : 1 }
```

2. Create a snapshot of each Amazon EBS volume you want to use to hold data, journals, or oplogs for `mongod`, using the volume numbers on your specific instance. You can find these numbers in the AWS Management Console or by using the `ec2-describe-instances` command.

```
$ ec2-create-snapshot -d thedescriptionofyourbackup vol-11111111
```

3. After the `ec2-create-snapshot` command returns the Pending state for the snapshot, it is safe to resume operations on `mongod`. Unlock the database and resume processing.

```
> db.$cmd.sys.unlock.findOne(); { "ok" : 1, "info" : "unlock requested" }
```

Note:: For the duration of a snapshot (until it is listed as "Completed"), there will be a variable negative impact to Amazon Elastic Block Store disk performance. If you are operating near capacity, you should take your system snapshots on a time-delayed replica.

Restore

To restore from a backup

1. Create an Amazon EBS volume from each snapshot used to back up your data. This step could require more than one volume. You can find the snapshot IDs and create the volumes by using the AWS Management Console or by using the `ec2-describe-snapshots` command.

```
$ ec2-create-volume --availability-zone us-east-1a --snapshot snap-12345678
```

2. Attach the volumes to the instance. Remember, if you're restoring a RAID set, to replace the volumes in the same order for easiest re-creation of the RAID volume in the OS.

```
$ ec2-attach-volume --instance i-aaaaaaa --device /dev/sdp vol-01010101
```

3. Make a directory into which you can mount the file system.

```
$ sudo mkdir -p /data/db/

$ sudo chown `id -u` /data/db
```

4. Mount the volume.

```
$ sudo mount /dev/sdp /data/db
```

5. Start `mongod` and then verify the restoration of your MongoDB collections.

```
$ service start mongod
> db.thenameofyourcollection.validate({full:true})
```

Monitoring

The Amazon CloudWatch service provides robust monitoring of Amazon EC2 instances, Amazon EBS volumes, and other services. Amazon CloudWatch can send an alarm by SMS or email when user-defined thresholds are reached on individual Amazon Web Services. For example, you can set an alarm to warn of excessive storage throughput. For information, go to Send Email Based on Storage Throughput Alarm in the *Amazon CloudWatch Developer Guide*. Alternatively, you can write a custom metric to Amazon CloudWatch, for example, current free memory on your instances, and to set alarms or trigger automatic responses when those measures exceed a threshold that you specify.

10gen, the company behind MongoDB, has also released MongoDB Monitoring Service (MMS), a free, cloud-based product for monitoring and managing MongoDB installations. More details on this service are available at http://www.10gen.com/mongodb-monitoring-service.

Security

Network

To access MongoDB resources from other Amazon EC2 instances or from remote servers, you need to configure your security group to permit ingress over the appropriate TCP ports. Like any network service, these ports should be opened conservatively; for example, open them only to your corporate office or to other authenticated and authorized machines that require access.

Default TCP port numbers for MongoDB processes are as follows:

A standalone `mongod` process: 27017
`mongos` : 27017
`shard server (mongod --shardsvr)` : 27018
`config server (mongod --configsvr)` : 27019
web statistics page for `mongod` : add 1000 to port number (28017, by default).

Most statistics pages in the HTTP UI are unavailable unless the --rest option is specified. To disable the "home" stats page, use `--nohttpinterface`. (If you use the 28017 port, you should secure it. In any case, the information on the stats home page is read-only.)

Ports can be opened to all computers in the same Amazon EC2 security group, to another security group, to a specific IP address, or to a CIDR IP address range. Security group parameters can be changed at any time. For extra security, some users use automation scripts to open these ports when needed and close them again when MongoDB resources are not in use.

The best practice in a multitier architecture is to permit operational access to the `mongod` tier only from servers in the security groups that require access and to control access only from known administrative IP addresses.

Authentication

To enable authentication, run the `mongod` process with the `--auth` option. You need to add a user to the `admin` collection before you start the server, or else add the first user from the `localhost` interface.

```
> use admin > db.addUser("theadmin", "anadminpassword")
```

To authenticate a replicated or sharded mongod installation

1. Create a key file (Base64, 6 or more characters, no larger than 1 KB) that can be copied to each server in the set.
2. Modify this file's permissions to be readable only by the appropriate user.
3. Start each server in the cluster, including all replica set members, all `configservers`, and all `mongos` processes, with the `--keyFile /path/to/file` option.

Conclusion

The AWS cloud provides a unique platform for any NoSQL application, including MongoDB. With capacities that can meet dynamic needs, cost that is based on use, and easy integration with other AWS products like Amazon CloudWatch, CloudFormation, and Identity and Access Management, the AWS cloud enables you to run a variety of NoSQL applications without having to manage the hardware yourself.

MongoDB, in combination with AWS, provides a robust platform for developing scalable, high-performance applications.

For more information on Amazon AWS, see http://aws.amazon.com.

Further Reading/Next Steps

- MongoDB on AWS Presentation http://www.10gen.com/presentations/mongosf-2011/running-mongodb-cloud
- MongoHQ – Hosted MongoDB Platform running on AWS http://www.mongohq.com/
- MongoDB Best Practices by EngineYard http://www.engineyard.com/blog/2011/mongodb-best-practices/
- Getting Started with MongoDB and AWS http://www.mongodb.org/display/DOCS/Amazon+EC2

Document Revisions

March 2013

- Added guidance on Provisioned IOPS EBS
- Added guidance on cr1.8xlarge, hi1.4xlarge
- Added enumeration of MongoDB IO sources
- Altered installation instructions to use a package manager and eliminate unnecessary configuration options
- Eliminated characterization that replication provides enhanced read performance.
- Improved guidance on HA installations for 2-AZ Regions.
- Provided guidance on t1.micro instances
- Altered guidance on storage configuration to use LVM on MDADM over distinct RAID sets.
- Updated recommended MongoDB version.
- Updated scripts to reflect EBS-Optimized and pIOPS flags.

January 2012

- Initial release