Apache Cassandra on AWS

Guidelines and Best Practices

January 2016

Notices

This document is provided for informational purposes only. It represents AWS's current product offerings and practices as of the date of issue of this document, which are subject to change without notice. Customers are responsible for making their own independent assessment of the information in this document and any use of AWS's products or services, each of which is provided "as is" without warranty of any kind, whether express or implied. This document does not create any warranties, representations, contractual commitments, conditions or assurances from AWS, its affiliates, suppliers or licensors. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Abstract

Amazon Web Services (AWS) is a flexible, cost-effective, easy-to-use cloud-computing platform. Apache Cassandra is a popular NoSQL database that is widely deployed in the AWS cloud. Running your own Cassandra deployment on Amazon Elastic Cloud Compute (Amazon EC2) is a great solution for users whose applications have high throughput requirements.

This whitepaper provides an overview of Cassandra and its implementation on the AWS cloud platform. It also talks about best practices and implementation characteristics such as performance, durability, and security, and focuses on AWS features relevant to Cassandra that help ensure scalability, high availability, and disaster recovery in a cost-effective manner.

Introduction

NoSQL databases are a type of database optimized for high-performance operations on large datasets. Each type of NoSQL database provides its own interface for accessing the system and its features. One way to choose a NoSQL database types is by looking at the underlying data model, as shown following:

- Key-value stores: Data is organized as key-value relationships and accessed by primary key. These products are typically distributed row stores. Examples are Cassandra and Amazon DynamoDB.
- Graph databases: Data is organized as graph data structures and accessed through semantic queries. Examples are Titan and Neo4J.
- Document databases: Data is organized as documents (for example, JSON files) and accessed by fields within the documents. Examples are MongoDB and DynamoDB.
- Columnar databases: Data is organized as sections of columns of data, rather than rows of data. Example: HBase.

DynamoDB shows up in both document and key-value stores in this list because it supports storing and querying both key-value pairs and objects in a document format like JSON, XML, or HTML.

NoSQL on AWS

Amazon Web Services provides several NoSQL database software options for customers looking for a fully managed solution, or for customers who want full control over their NoSQL databases but who don't want to manage hardware infrastructure. All our solutions offer flexible, pay-as-you-go pricing, so you can quickly and easily scale at a low cost.

Consider the following options as possible alternatives to building your own system with open source software (OSS) or a commercial NoSQL product.

- Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability.[1] All data items in DynamoDB are stored on solid-state drives (SSDs) and are automatically replicated across three facilities 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.

- Amazon Simple Storage Service (Amazon S3) provides a simple web services interface that can store and retrieve any amount of data anytime from anywhere on the web.[2] Amazon S3 gives developers access to the same highly scalable, reliable, secure, fast, and 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.

Cassandra: A Brief Introduction

Note: This is a brief overview on how Cassandra works; to learn more visit DataStax documentation.[3]

Apache Cassandra is a massively scalable open source NoSQL database, which is ideal for managing large amounts of structured, semi-structured, and unstructured data across multiple distributed locations. Cassandra is based on log-structured merge-tree, a data structure that is highly efficient with high-volume write operations.[4] The most popular use case for Cassandra is storing time series data.

Cassandra delivers continuous availability, linear scalability, and operational simplicity across many commodity servers with no single point of failure, along with a powerful dynamic data model designed for maximum flexibility and fast response times. Cassandra is a master less peer-to-peer distributed system where data is distributed among all nodes in the cluster. Each node has knowledge about the topology of the cluster and exchanges information across the cluster every second.

Cassandra: Key Terms and Concepts

Before we discuss best practices and considerations for using Cassandra on AWS, let us review some key concepts.

A *cluster* is the largest unit of deployment in Cassandra. Each cluster consists of nodes from one or more distributed locations (Availability Zones or AZ in AWS terms).

A *distributed location* contains a collection of nodes that are part of a cluster. In general, while designing a Cassandra cluster on AWS, we recommend that you use multiple Availability Zones to store your data in the cluster. You can configure Cassandra to replicate data across multiple Availability Zones, which will allow your database cluster to be highly available even during the event of an Availability Zone failure. To ensure even distribution of data, the number of Availability Zones should be a multiple of the replication factor. The Availability Zones are also connected through low-latency links, which further helps avoid latency for replication.

A *node* is a part of a single distributed location in a Cassandra cluster that stores partitions of data according to the partitioning algorithm.

A *commit log* is a write-ahead log on every node in the cluster. Every write operation made to Cassandra is first written sequentially to this append-only structure, which is then flushed from the write-back cache on the operating system (OS) to disk either periodically or in batches. In the event of a node recovery, the commit logs are replayed to perform recovery of data.

A *memtable* is basically a write-back cache of data rows that can be looked up by key. It is an in-memory structure. A single memtable only stores data for a single table and is flushed to disk either when node global memory thresholds have been reached, the commit log is full, or after a table level interval is reached.

An *SStable* (sorted string table) is a logical structure made up of multiple physical files on disk. An SStable is created when a memtable is flushed to disk. An SStable is an immutable data structure. Memtables are sorted by key and then written out sequentially to create an SStable. Thus, write operations in Cassandra are extremely fast, costing only a commit log append and an amortized sequential write operation for the flush.

A *bloom filter* is a probabilistic data structure for testing set membership that never produces a false negative, but can be tuned for false positives. Bloom filters are off-heap structures. Thus, if a bloom filter responds that a key is not present in an SStable, then the key is not present, but if it responds that the key is present in the SStable, it might or might not be present. Bloom filters can help scale read requests in Cassandra. Bloom filters can also save additional disk read operations reading the SStable, by indicating if a key is not present in the SStable.

An *index file* maintains the offset of keys into the main data file (SStable). Cassandra by default holds a sample of the index file in memory, which stores the offset for every 128th key in the main data file (this value is configurable). Index files can also help scale read operations better because they can provide you the random position in the SStable from which you can sequentially scan to get the data. Without the index files, you need to scan the whole SStable to retrieve data.

A *keyspace* is a logical container in a cluster that contains one or more tables. Replication strategy is typically defined at the keyspace level.

A *table*, also known as a *column family*, is a logical entity within a keyspace consisting of a collection of ordered columns fetched by row. Primary key definition is required while defining a table.

Write Request Flow

The following diagram shows a Cassandra cluster with seven nodes with a replication factor of 3. The clients are writing to the cluster using quorum consistency level.[5] While using quorum consistency level, write operations succeed if two out of three nodes acknowledge success to the coordinator (the node that the client connects to).

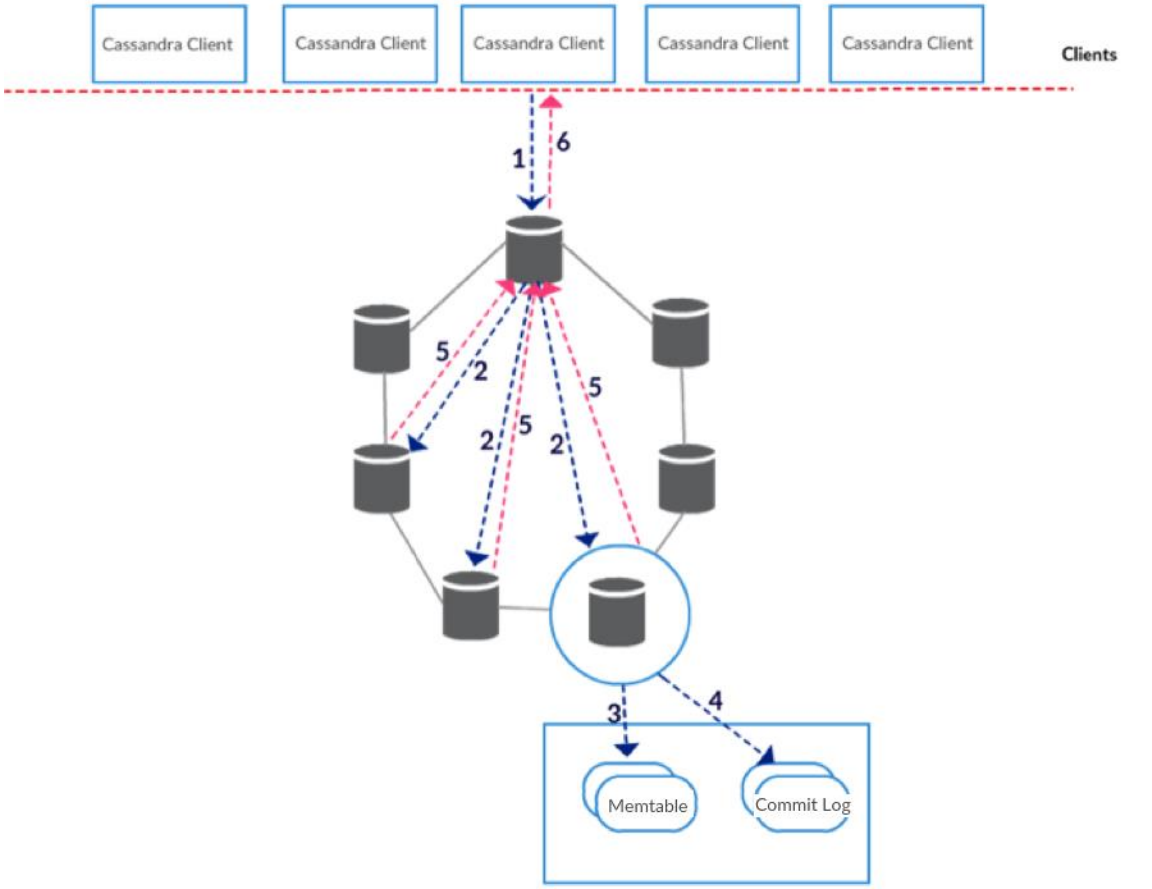

Figure 1: Write Request Flow

The preceding diagram illustrates a typical write request to Cassandra with three-way replication, as described following:

1. A client sends a request to a node in the cluster to store a given key. At this point, the node might or might not be the right partition to store the key. If it is not the right partition, the node acts as a coordinator (the case in this example). Note that a node can either act as a replica or a coordinator or both (if the node maps to the data and is talking to the client).
2. The coordinator determines the replica nodes that should store the key and forwards the request to those nodes.

3. Each node that gets the key performs a sequential write operation of the data, along with the metadata required to recreate the data in the commit log locally.
4. The key along with its data is written to the in-memory memtable locally.
5. Replica nodes respond back to the coordinator with a success or failure.
6. Depending on the consistency level specified as part of the request, the coordinator will respond with success or failure to the client. For example, with a consistency level of quorum and a replication factor of 3, the coordinator will respond with success as soon as two out of three nodes respond with success.

Now, during step 5 preceding, if some nodes do not respond back and fail (for example, one out of three nodes), then the coordinator stores a hint locally to send the write operation to the failed node or nodes when the node or nodes are available again. These hints are stored with a time to live equal to the *gc_grace_seconds* parameter value, so that they do not get replayed later. Hints will only be recorded for a period equal to the *max_hint_window_in_ms* parameter (defined in cassandra.yaml), which defaults to three hours.

As the clients keep writing to the cluster, a background thread keeps checking the size of all current memtables. If the thread determines that either the node global memory thresholds have been reached, the commit log is full, or a table level interval has been reached, it creates a new memtable to replace the current one and marks the replaced memtable for flushing. The memtables marked for flush are flushed to disk by another thread (typically, by multiple threads).

Once a memtable is flushed to disk, all entries for the keys corresponding to that memtable that reside in a commit log are no longer required, and those commit log segments are marked for recycling.

When a memtable is flushed to disk, a couple of other data structures are created: a bloom filter and an index file.

Compaction

The number of SStables can increase over a period of time. To keep the SStables manageable, Cassandra automatically performs minor compactions by default. Compaction merges multiple SStables based on an algorithm that you specify using a compaction strategy.

Compaction allows you to optimize your read operations by allowing you to read a smaller number of SStables to satisfy the read request. Compaction basically merges multiple SStables based on the configurable threshold to create one or more new, immutable SStables. For example, the default compaction strategy, Size Tiered Compaction, groups multiple similar-sized SStables together and creates a single large SStable. It keeps iterating this process on similar-sized SStables.

Compaction does not modify existing SStables (remember, SStables are immutable) and only creates a new SStable from the existing ones. When a new SStable is created, the older ones are marked for deletion. Thus, the used space is temporarily higher during compaction. The amount of space overhead due to compaction depends on the compaction strategy used. This space overhead needs to be accounted for during the planning process. SStables that are marked for deletion are deleted using a reference counting mechanism or during a restart.

Read Request Flow

Before we dive into the read request flow, we will summarize what we know about a Cassandra cluster.

In a cluster, each row is replicated across multiple nodes (depending on your replication factor). There is no concept of a master node. This approach means that any node in the cluster that contains the row can answer queries about that row. Cassandra uses the Gossip protocol to exchange information about network topology among nodes. By virtue of Gossip, every node learns about the topology of the cluster and can determine where a request for a given row should be sent to in the cluster.

In the diagram following, we have a Cassandra cluster with seven nodes and a replication factor of 3. The clients read from the cluster using quorum consistency level. While using quorum consistency level, read operations succeed if two out of three nodes acknowledge success.

With this brief context, let us look at how the read requests are served. The figure and list following illustrate.

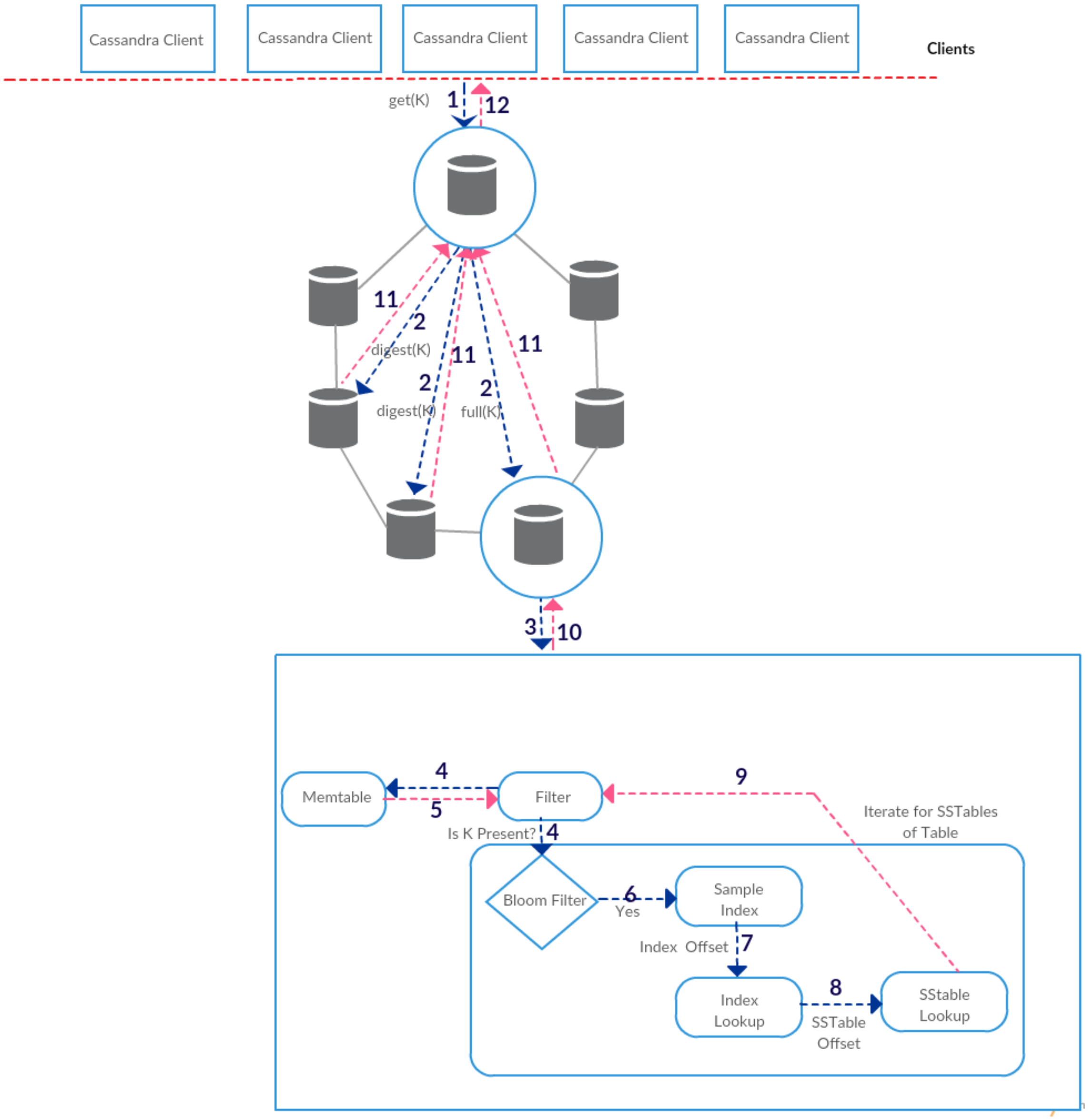

Figure 2: Read Request Flow

1. A client sends a request to a node in the cluster to get data for a given key, K. At this point, if the key is not mapped to this node, then the node acts as

a coordinator. Note that a node can either act as a replica or a coordinator or both (if the node maps to the data and is talking to the client).

2. The coordinator determines the replica nodes that might contain the key and forwards the request to those nodes. While sending the request to the replica nodes, the coordinator determines which node is closer to itself (through a snitch) and sends a request for full data to the closest node and a request for the digest generated with the hash of the data from the other nodes. (A *snitch* determines which host is closest to the current location.)
3. The request is forwarded to the internal services of the node for further processing.
4. A request for data from both the memtable and SStables is made. The request iterates over the bloom filters for the SStables asking whether the key is present.
5. Because the memtable is in memory, data might be returned faster from the memtable, but one or more SStables still need to be consulted for the data.
6. If a bloom filter responds that the key is not present, the next bloom filter is checked. If a bloom filter responds that the key might be present (which is the case here), then it checks the sample index in memory.
7. A binary search is performed on the sample index to determine a starting offset into the actual index file. This offset is used to offset into the index file and do a sequential read operation to obtain the offset into the SStable for the actual key.
8. With the offset obtained from step 7, the actual data from the SStable is returned by offsetting into the SStable file.
9. The data for the key is returned from the SStable lookup. The filter command consolidates all versions of the key data obtained from SStable lookups and the memtable.
10. The latest consolidated version of the key data is returned to the internal services.
11. The same process is repeated by the internal services on other nodes and results are returned back to the coordinator node.

12. The coordinator compares the digest obtained from all nodes and determines if there is a conflict within the data. If there is a conflict, the coordinator reconciles the data and returns the reconciled version back to the client. A read repair is also initiated to make the data consistent.

Note that we did not talk about the cache preceding. To learn more about caching, refer to the DataStax documentation.[6]

Read repairs can resolve data inconsistencies when the written data is read. But when the written data is not read, you can only use either the hinted handoff [7] or anti-entropy [8] mechanism.

Cassandra: Resource Requirements

Let us now take a look at the resources required to run Cassandra. We will look at storage and I/O, CPU, memory, and networking requirements.

Storage and IO Requirements

Most of the I/O happening in Cassandra is sequential. But there are cases where you require random I/O. An example is when reading SStables during read operations.

SSD is the recommended storage mechanism for Cassandra, because it provides extremely low-latency response times for random read operations while supplying ample sequential write performance for compaction operations.

Replication and storage overhead due to compaction has to be taken into account while determining storage requirements.

The recommended file system for all volumes is XFS. Ext4 might be used by preference. Ext3 is considerably slower, and we recommend that you avoid it.

AWS provides two types of storage options, namely local storage and Amazon Elastic Block Store (Amazon EBS). Local storage is available locally to the instance, and EBS is network-attached storage. We will talk more about choosing a storage option on AWS for Cassandra later in this whitepaper.

Network Requirements

Cassandra uses the Gossip protocol to exchange information with other nodes about network topology. The use of Gossip coupled with distributed nature of Cassandra, which involves talking to multiple nodes for read and write operations, results in a lot of data transfer through the network.

That being the case, we recommend to always choose instances with at least 1 Gbps network bandwidth to accommodate replication and Gossip. When using AWS, we recommend choosing instance types with enhanced networking enabled. Enhanced networking offers better network performance. We will discuss more about this option and its benefits later in this whitepaper.

Memory Requirements

Cassandra basically runs on a Java virtual machine (JVM). The JVM has to be appropriately sized for performance. Large heaps can introduce garbage collection (GC) pauses that can lead to latency, or even make a Cassandra node appear to have gone offline. Proper heap settings can minimize the impact of GC in the JVM.

The *MAX_HEAP_SIZE* parameter determines the heap size of the Cassandra JVM. DataStax recommends not to allocate more than 8 GB for the heap.

The *HEAP_NEW_SIZE* parameter is the size of the young generation in Java. A general rule of thumb is to set this value at 100 MB per vCPU on Amazon EC2.

Cassandra also largely depends on the OS file cache for read performance. Hence, choosing an optimum JVM heap size and leaving enough memory for OS file cache is important. For production workloads, our general recommendation is to choose an instance type with at least 32 GB of DRAM.

Learn more about JVM tuning in the DataStax documentation.[9]

CPU Requirements

When looking at Cassandra CPU requirements, it's useful to note that insert-heavy workloads are CPU-bound in Cassandra before becoming IO-bound. In other words, all write operations go to the commit log, but Cassandra is so efficient in writing, that the CPU becomes the limiting factor. Cassandra is highly concurrent and uses as many CPU cores as available.

When choosing instance types with Amazon EC2 for write-heavy workloads, you should look at instance types with at least 4 vCPUs. Although this is a good starting point, we recommend that you test a representative workload before settling on the instance type for production.

Planning Cassandra Clusters on AWS

This section discusses how you can apply Cassandra features to AWS services to deploy Cassandra in the most optimal and efficient way.

Planning Regions and Availability Zones

The AWS Cloud Infrastructure is built on Regions and Availability Zones (“AZs”). [10]A *Region* is a physical location in the world where we have multiple Availability Zones. *Availability Zones* consist of one or more discrete data centers, each with redundant power, networking and connectivity, housed in separate facilities. These Availability Zones offer you the ability to operate production applications and databases which are more highly available, fault tolerant and scalable than would be possible from a single data center (*learn more*). This also helps customers implement regulatory compliance and geographical expansion.

You can use AWS’ global infrastructure to manage network latency and to address your regulatory compliance needs. For example, data in one region is not automatically replicated outside that region. If your business requires higher availability, it is your responsibility to replicate data across regions.

When you are building your Cassandra cluster, select the same region for your data and application to minimize application latency. For details on the exact location of AWS regions, visit the Region Table. [11]

Unlike legacy master-slave architectures, Cassandra has a master less architecture in which all nodes play an identical role, so there is no single point of failure. Consider spreading Cassandra nodes across multiple Availability Zones to enable high availability. By spreading nodes across Availability Zones, in the case of a disaster, you can still maintain availability and uptime.

Cassandra clusters can be made Amazon EC2–aware and thus support high availability by defining an appropriate snitch setting through the *endpoint_snitch* parameter in cassandra.yaml.[12] This parameter can be either set to Ec2Snitch[13] or Ec2MultiRegionSnitch.[14] Ec2Snitch allows you to build clusters within a single region, and Ec2MultiRegionSnitch allows you to deploy clusters in multiple AWS regions.

Setting up a snitch allows Cassandra to place the replicas for data partitions on nodes that are in different Availability Zones. Thus, if your keyspace has a replication factor of 3 in the US East (N. Virginia) region, us-east-1, then the Ec2Snitch setting allows your data to be replicated across three Availability Zones in us-east-1. For example, if you set up a cluster in Availability Zones us-east-1a, us-east-1b, and us-east-1c and set a replication factor of 3 at the keyspace level, then each write operation will be replicated across nodes in these three Availability Zones.

Basic Tip
If you are just starting out with Cassandra, plan for cluster growth from the beginning. Choose Ec2MultiRegionSnitch to avoid complications when you decide to expand your cluster.

The snitch setting thus provides higher availability by allowing Cassandra to place replicas of your data in different Availability Zones when a write operation happens.

Cassandra also has *seed nodes,* which are initially consulted by a new node that wants to bootstrap and join the Cassandra ring. In the absence of a seed node, bootstrapping does not happen and the node does not start. We highly recommend that you spread your seed nodes across multiple Availability Zones. In this situation, even if an Availability Zone goes down, the new nodes can bootstrap from the seed nodes from another Availability Zone.

Cassandra clusters can also be made datacenter-aware and rack-aware. “Datacenter” in Cassandra terminology translates to a “region” in AWS terms, and “rack” translates to an “Availability Zone” in AWS terms.

Cassandra clusters can be designed to be highly resilient to failures by leveraging regions and Availability Zones in AWS coupled with the EC2 snitches that come packaged with the software. For example, with a three-AZ design, requests can still continue to succeed in the event of an entire Availability Zone failure. You can create live backups of your cluster by designing a cluster in another AWS region and let Cassandra handle the asynchronous replication.

Planning an Amazon Virtual Private Cloud

Amazon Virtual Private Cloud (Amazon VPC) lets you provision a logically isolated section of the AWS cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.

We highly recommend that you launch your Cassandra cluster within a VPC. One of the biggest benefits that VPC offers for Cassandra workloads is the enhanced networking feature. Enabling enhanced networking on your instance results in higher performance (more packets per second), lower latency, and lower jitter. At the current time, C3, C4, D2, I2, M4 and R3 are the instance families for which enhanced networking is supported within a VPC.[15] In order to enable this feature, you must also launch a hardware virtual machine (HVM) AMI with the appropriate SR-IOV driver.

Best practices for setting up Cassandra on VPC

First, create a large enough VPC with a /16 Classless Inter-Domain Routing (CIDR) block (for example, 10.0.0.0/16) to accommodate a sufficient number of instances within a single VPC. This approach does not impact performance in any way, and it gives you the room to scale when needed. If you instead create a smaller VPC, for example 10.0.0.0/28, which has up to 14 IP addresses, then you have to create a new VPC. Cassandra clusters scale with data by throwing more capacity at it, and having a large enough VPC will make it easier to scale.

You can define both public and private subnets within a VPC. We recommend that you host your Cassandra clusters in a private subnet within your VPC, which does not have Internet access. You can then set up a Network Address Translation (NAT) instance in a public subnet to allow the cluster nodes to talk to the Internet for software updates.

Because subnets do not span across Availability Zones, plan to create multiple subnets, depending on the replication factor you intend to configure for your Cassandra keyspace.[16] For example, if your replication factor is 3, you can create three subnets, each in a different Availability Zone, to host your cluster. Another thing to account for while planning subnets for your Cassandra cluster is that Amazon reserves the first four IP addresses and the last IP address of every subnet for IP networking purposes.

If you have administrators who need Secure Shell (SSH) access to the nodes in the cluster for maintenance and administration, the industry standard practice is to configure a separate bastion host.

If you already have a Cassandra cluster with AWS, but on the EC2-Classic platform, we recommend migrating the cluster over to VPC to leverage the higher performance features and enhanced security offered by VPC. To make this migration simple, AWS offers ClassicLink, which allows you to enable communication between EC2-Classic and VPC and to incrementally migrate your Cassandra cluster to VPC with no downtime.[17] We will discuss more about this feature in the Migration section.

Planning Elastic Network Interfaces

An *elastic network interface (ENI)* is a virtual network interface that you can attach to an instance in a VPC in a single Availability Zone. You can create an ENI, attach it to an instance, detach it from that instance, and attach it to another instance in the same Availability Zone. When you do so, the attributes of the ENI follow as it is detached and reattached. When you move an ENI from one instance to another, network traffic is redirected to the new instance. The maximum number of ENIs you can attach per instance depends on the instance type, as discussed in Elastic Network Interfaces (ENI) in the *Amazon EC2 User Guide.*[18]

When working with Cassandra, ENIs are generally used to support seed node configuration. Seed node IP addresses are hard-coded in the Cassandra .yaml configuration file. If a seed node fails, the yaml configuration on every node in the cluster needs to be updated with the IP address of the new seed node that is brought up in place of the failed seed node. This could create operational difficulties with large clusters.

However, you can avoid this scenario if you attach an ENI to each seed node and add the ENI IP address to the list of seed nodes in the .yaml configuration file. If you do so, in the event of failure of a seed node, you can automate in such a way that the new seed node takes over the ENI IP address programmatically. The automation would typically involve using the AWS CLI and running ENI specific commands to perform the detach and attach.

Note that attaching an ENI to an EC2 instance does not increase the network bandwidth available to the instance.

Planning High-Performance Storage Options

Understanding the total I/O required is key to selecting an appropriate storage configuration on AWS. Most of the I/O driven by Cassandra is sequential for a write-heavy workload. However, read-heavy workloads require random access, and if your working set does not fit into memory, your setup will likely become I/O bound at some point. Hence, making a good decision initially when you are choosing your storage is important.

AWS offers two main choices to construct the storage layer of your Cassandra infrastructure: Amazon EBS volumes and Amazon EC2 instance stores.

Amazon Elastic Block Store (EBS) Volumes

Amazon EBS provides persistent block-level storage volumes for use with Amazon EC2 instances in the AWS cloud. Each Amazon EBS volume is automatically replicated within its Availability Zone to protect you from component failure, offering high availability and durability.

Amazon EBS volumes offer the consistent and low-latency performance needed to run your workloads. Amazon EBS volumes provide a great design for systems that require storage performance variability.

There are two types of Amazon EBS volumes you should consider for Cassandra deployments:

- General Purpose (SSD) volumes offer single-digit millisecond latencies, deliver a consistent baseline performance of 3 IOPS/GB to a maximum of 10,000 IOPS, and provide up to 160 MB/s of throughput per volume.[19]

- Provisioned IOPS (SSD) volumes offer single-digit millisecond latencies, deliver a consistent baseline performance of up to 30 IOPS/GB to a maximum of 20,000 IOPS, and provide up to 320 MB/s of throughput per volume. These features making it much easier to predict the expected performance of a system configuration.[20]

DataStax recommends both these EBS volume types for Cassandra workloads. For more information, see the DataStax documentation.[21] The magnetic volume types are generally not recommended for performance reasons.

At minimum, using a single EBS volume on an Amazon EC2 instance can achieve 10,000 IOPS or 20,000 IOPS from the underlying storage, depending upon the volume type (GP2 or PIOPS). For best performance, use Amazon EBS–optimized instances.[22]

Tip
DataStax recommends GP2 and Provisioned IOPS EBS volume types for Cassandra workloads. For more information, see the DataStax documentation

An Amazon EBS–optimized instance uses an optimized configuration stack and provides additional, dedicated capacity for Amazon EBS I/O. This optimization provides the best performance for your EBS volumes by minimizing contention between Amazon EBS I/O and other traffic from your instance.

EBS-optimized instances deliver dedicated throughput between Amazon EC2 and Amazon EBS, with options between 500 and 4,000 megabits per second (Mbps) depending on the instance type used. For example, an EBS-optimized instance in a Cassandra setup can provide data volume at 4 TB and 10,000 IOPS with a commit log at 1 TB and 3000 IOPS.

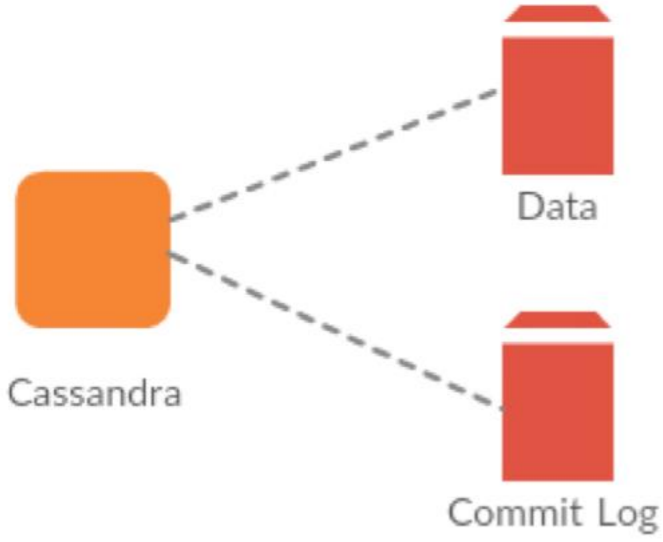

Figure 3: Single EBS Volume

To scale IOPS further beyond that offered by a single volume, you can use multiple EBS volumes, as shown following.

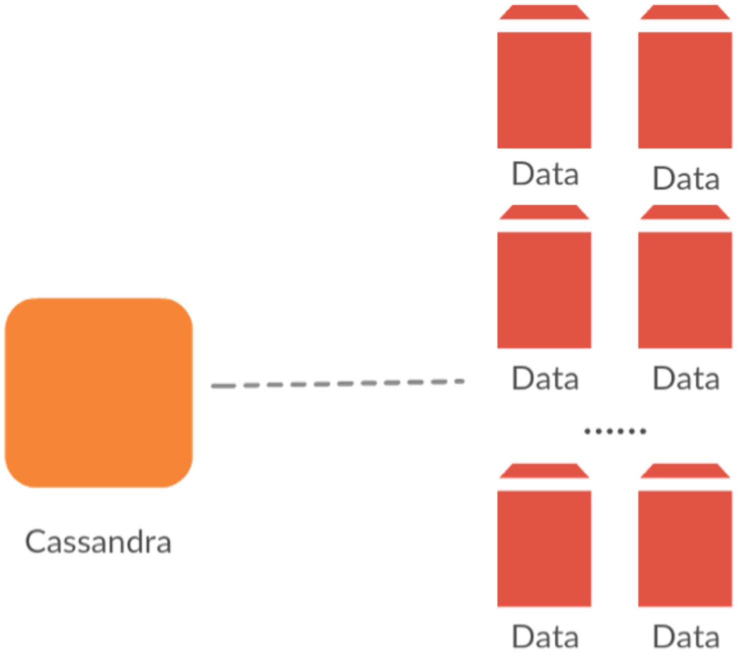

Figure 4: Multiple EBS Volumes for Data

You can choose from multiple combinations of volume size and IOPS, but remember to optimize based on the maximum IOPS supported by the instance.

In the configuration just shown, you might want to attach volumes with combined IOPS beyond the IOPS offered by the EBS-optimized EC2 instance. For example, one Provisioned IOPS (SSD) volume with 16,000 IOPS or two General Purpose (SSD) volumes with 8,000 IOPS striped together matches the 16,000 IOPS offered by a c4.4xlarge instance through its EBS-optimized dedicated bandwidth.

Instances with a 10 Gbps network and enhanced networking can provide up to 48,000 IOPS and 800 MB/s of throughput to Amazon EBS volumes. For example, with these instances, five General Purpose (SSD) volumes of 10,000 IOPS each can saturate the link to Amazon EBS.

You should also note that EBS-optimized connections are full duplex, and can drive more throughput and IOPS in a 50/50 read/write workload where both communication lanes are used.

Best Practice Tip
The general best practice when using EBS volumes for storage is to use separate volumes for the commit log and data. This approach will allow you to scale better.

The best practice while using EBS volumes for Cassandra is to use separate volumes for commit log and data directories. This approach will allow you to scale better.

Amazon EBS also provides a feature for backing up the data on your EBS volumes to Amazon S3 by taking point-in-time snapshots.

Amazon EC2 Instance Stores

Many Amazon EC2 instance types can access disk storage located on disks that are physically attached to the host computer. This disk storage is referred to as an *instance store*. If you're using an instance store on instances that expose more than a single volume, you can stripe the instance store volumes (using RAID0) to enhance I/O throughput. Remember, if the instance is stopped, fails, or is terminated, you'll lose all your data. Therefore, we strongly recommend setting up replication across multiple instances across Availability Zones for your Cassandra cluster.

When using a logical volume manager (for example, mdadm or LVM), make sure that all metadata and data are consistent when you perform the backup.

Planning Instance Types Based on Storage Needs

Before we go into instance type recommendations, let us first go over an example to understand the storage and compute requirements to handle a Cassandra workload. Let us assume that we have a single table (column family) for a cluster to keep things simple.

Assumptions and Storage Formula

We will make the following assumptions for this example:

- 100 columns per row.
- 10,000 write operations per second and 500 read operations per second (95 percent writes and 5 percent reads).
- For the sake of simplicity, we will also assume that the reads and writes per second is uniform throughout the day.
- The total size of each column name and column value is 30 bytes.
- We are storing time series data, and hence the columns are all set to expire. We set the expiration to 1 month because we do not need data older than a month in our table.
- The average size of a primary key is 10 bytes.
- The replication factor is 3.
- The compaction strategy is size-tiered (50 percent storage overhead).

Let us calculate the storage requirements for one month for this example:

```
Storage requirement = (((Number_of_columns * (column_name_size
+ column_value_size + 23)) + 23) * Number_of_rows +
Number_of_rows * (32 + primary_key_size)) * Replication_Factor
* (1 + Compaction_Overhead)/1024/1024/1024/1024 TB
```

This formula can in general be applied to calculate your storage requirements at a column family level.

Note also these important points:

- The storage overhead per column is 23 bytes for a column that expires, as with time series.
- The storage overhead per row is 23 bytes.
- The storage overhead per key for the primary key is 32 bytes.
- For a regular column, the storage overhead is 15 bytes.
- The total rows written per month are as follows:
  `10,000*86400*30 = 25920000000`

Here are our variables:

- `Number_of_columns = 100`
- `column_name_size = 20`
- `column_value_size = 10`
- `Number_of_rows = 25920000000`
- `Primary_key_size = 10`
- `Replication_Factor = 3`
- `Compaction_Overhead = 50 percent(0.5)`

Applying the preceding formula with the values from our example preceding, the storage requirement equals 569 TB. Thus, we will need at least 569 TB of disk space to provision for this cluster.

Let us now compare our options available on AWS to satisfy this requirement.

Using I2 Instances

The most common instance type that customers use for Cassandra on AWS is the I2 instance. I2 instances with high I/O are optimized to deliver tens of thousands of low-latency, random IOPS to applications. The I2 instance type offers enough disk, IOPS, and memory to allow you to run a low latency workload for both read and write operations. I2 also supports enhanced networking. This instance type tends to get more frequently used when there are large amounts of data (in the range of terabytes) and IOPS (in the multiple tens of thousands) involved.

Looking at the I2 specifications, assuming use of the US East (N. Virginia) us-east-1 region, and accommodating 10 percent overhead for disk formatting, we can determine that for the storage we discussed previously, we will need 405 I2 instances (`569/((2*800*.9)/1024)`).[23] This workload will cost you $497,178 (`405*1.705*720`) for a month to run. This figure does not include data transfer charges, which are billed separately. We are also not including any calculations for commitlog for simplicity and assume that the data and commitlog will be collocated on the same drives.

However, do we really need 405 nodes for processing 10,000 write operations and 500 read operations per second on the I2? The answer might be no. In fact, we might be significantly overprovisioning every compute resource available to us other than local storage. We can move to a larger I2 instance with more local storage. But that might not help with cost. What might be an alternative option here?

If we can somehow decouple storage from compute, then we will have a viable scenario where we can provision optimum storage and compute power based on what is required. We can do this decoupling with EBS. The idea here is to use EBS to achieve the same or equivalent level of performance with lesser cost.

Decoupling with EBS

To perform this decoupling, we can provision EBS GP2 volumes, which can give us 3 IOPS/GB provisioned. Let us follow the DataStax recommendation for maximum data per node for Cassandra (3 to 5 TB) and assign 4 TB EBS GP2 volumes per node. This setup will give us 10,000 IOPS.

We also need to allocate storage for our commit logs, because keeping them on separate volumes provides better performance. We will allocate a 500 GB volume for the commit log on each node. This allocation will give about 1500 IOPS guaranteed, with a burst capability of up to 3000 IOPS for 30 minutes, depending on how much IOPS was saved earlier.

Now, we have to choose an instance type that allows enhanced networking and EBS-optimized bandwidth. There is an additional charge for EBS-optimized instances on many instance types, but AWS has recently launched new instance types that are EBS-optimized by default, with no extra cost for EBS-optimized bandwidth.

Using C4 Instances with EBS

The C4 instance is becoming a popular choice for running Cassandra on EBS. For an example, see the Crowd Strike presentation from re:Invent 2015.[24] C4 does not have any local storage and hence EBS must be attached for storage. In this specific case, where we have 95 percent write operations, C4 might be a great fit because you normally run out of CPU before you run out of IO for these write-skewed workloads.

We will also need an instance type in C4 that supports at least 3000 IOPS for the normal write throughput, plus compaction IOPS through its EBS-optimized bandwidth. Remember that we also recommended a minimum memory requirement of 32 GB of DRAM for production workloads. C4.4X fits into the CPU, memory, and IOPS requirements just described.

If we price this option, we require 143 C4.4X instances (4 TB per node for a total of 569 TB). We will get 572 TB (`143*4` TB).

Now, we need 500 GB of commit logs per node. Thus, we will require 72 TB GP2 just for hosting commit logs. Adding both data and commit logs (572 TB + 72 TB), we will need to provision 644 TB for EBS GP2.

Calculating EC2 costs for 143 C4.4XL instances, assuming the US East (N. Virginia) us-east-1 region, and using the current pricing, we come up with the following:[25]

```
143 C4.4X = 143*0.838*720 = $86,280.48
```

Calculating EBS volume costs, we come up with the following:

```
644 TB EBS GP2 = $65,946
```

Our total cost is EC2 cost plus EBS cost, which equals $152,226.

Comparing this with the cost of running I2, $497,178, we find that the C4.4X plus EBS option is about 3.2 times less expensive compared to the I2 option, without compromising on performance and reliability.

Cost Optimization for C4 Plus EBS and for I2

Now, when you settle on instance types for your Cassandra cluster, AWS offers pricing options such as Reserved Instances to further optimize your costs.

For example, let's say you decide to run the C4.4X plus EBS configuration for your cluster and you are happy with the cluster performance over a couple of months. At this point, with a stable instance choice for your cluster, it might be a good idea to make reservations and optimize costs. If you decide to make a three-year partial upfront reservation, it will save you 59 percent over on demand.

Here is the revised calculation:

```
143 C4.4X = 143*0.3476*720 = $35,788.89
```

```
644 TB EBS GP2 = $65,946
```

Our total cost now is $101,734.89. Thus, you can bring your cost down from $152,226 to $101,734.89 with a three-year reservation.

Now that we know the best optimized cost for the C4.4X plus EBS GP2 configuration, let us revisit the I2 option and apply reserved pricing to see how close we can get.

With three-year partial up-front on I2, you can save 74 percent over on demand, which means you only pay 26 percent over the on-demand cost. The cost drops to $129,266 (`$497,178/4`). Now the comparison between C4 option and the I2 option does not look too bad, but the C4 option is still about 1.2 times less expensive than the I2 option.

Summary for Example workload

Writes/Sec	Reads/Sec	Replication Factor	Storage(TB)	Avg.Row Size(KB)	Instance Count/Type	Cost
10000	500	3	570	3	405/I2	$497,178
10000	500	3	570	3	405/I2/3-Yr RI	$129,266
10000	500	3	644	3	143/C4.4XL	$152,226
10000	500	3	644	3	143/C4.4XL/3-Yr RI	$101,735

Using R3 or I2 Instances with Read-Heavy Workloads

If you are running a read-heavy workload, then you should look at using either the R3 instance family with or without EBS storage, or the I2 instance family. Cassandra depends heavily on the operating system's file system cache for read performance, and R3 instances deliver high sustained memory bandwidth with low network latency and jitter.

R3 instances come by default with attached local storage. If you find that your storage requirements are small enough after considering replication and storage overhead, then you might not need EBS for storage. If you do plan on using R3 with EBS, you should note that R3 instance types are not EBS-optimized by default and that there is an additional hourly cost when EBS optimization is enabled for R3 instances.

As an example of using R3 instances, suppose you want to store a total of 1 TB of data and have a high read throughput requirement. Given these factors, you can provision four r3.4xl, depending on your load tests. This approach costs you about $4000 per month. In this case, you do not want to reduce the number of nodes for availability reasons.

If you do not require the high memory and CPU offered with r3.4xl but still require 1 TB for storage and want to optimize this cost further, you can use four i2.x nodes and reduce the cost to about $2500.

For another example, suppose you want to store 6 TB of data and have a high read throughput requirement. In this case, you can provision 20 r3.4xl instances, which costs you about $20,000. However, let's assume you performed load tests and determined that from a compute perspective, six r3.4xl instances are more than enough to handle your workload. You also determine that most of the workload is memory bound because of the amount of DRAM available on r3.4x instances and that you will need a maximum of 2000 IOPS per instance. At this point, you can leverage EBS GP2 volumes and provision 1 TB of GP2 per instance. Doing this will bring down the number of nodes from 20 to 6 and bring your costs down to about $7000, which is about one-third the cost of using local storage for the same workload.

In general, if you are looking to drive over 20,000 IOPS per node for your Cassandra cluster, I2 instance types might be a better fit.

Horizontally Scaling Cassandra with EBS

As an additional note, if your workload requires pushing the disk bandwidth over the maximum EBS-optimized bandwidth for a specific instance type, you should leverage the horizontally scalable distributed nature of Cassandra and add more nodes to spread out the load. Spreading the load in this way allows your cluster to scale to virtually any amount of IOPS while using EBS at reasonable cost

Deploying Cassandra on AWS

Cassandra provides native replication capabilities for high availability and can scale horizontally, as the following illustration shows.

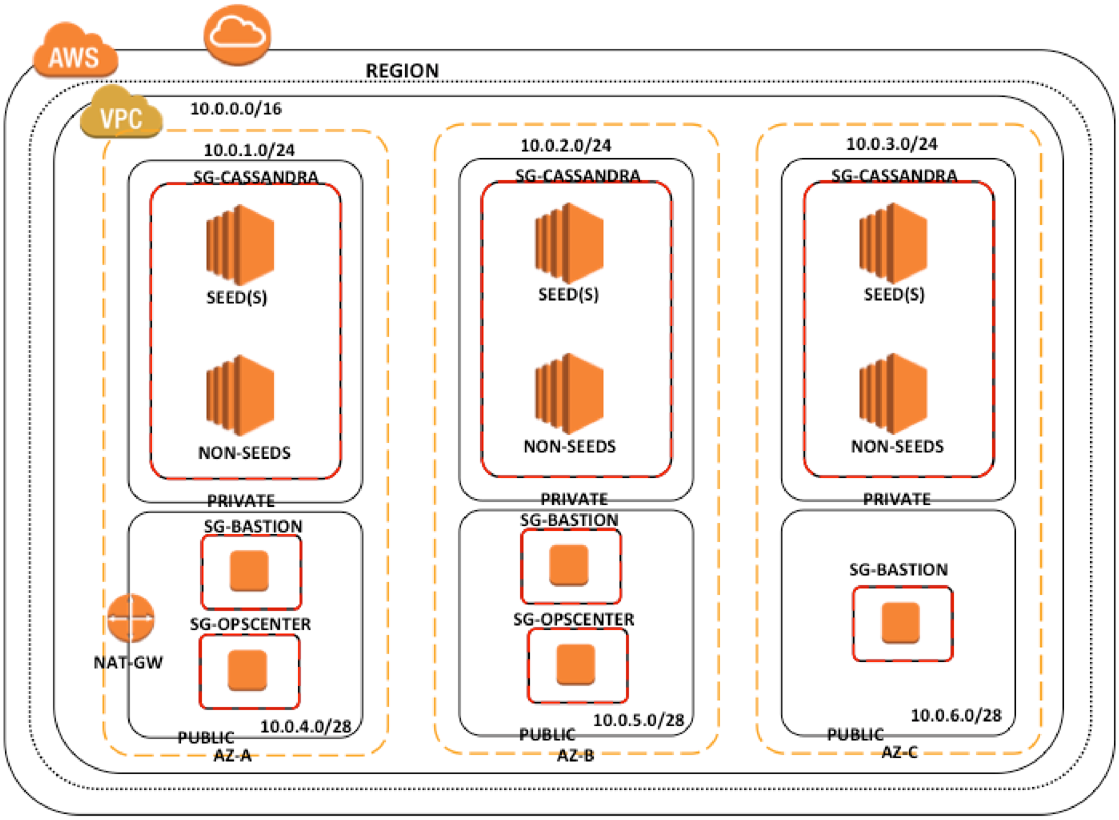

Figure 5: Highly Available Cassandra Cluster

Although you can scale vertically by using high-performance instances instead of a replicated and sharded topology, vertically scaled instances don't provide the significant fault tolerance benefits that come with a replicated topology. Because AWS has a virtually unlimited pool of resources, it is often better to scale horizontally.

Setting Up High Availability

The preceding architecture diagram shows a basic high-availability configuration for a Cassandra cluster. We highly recommend this setup for high availability. The cluster is in a large VPC with a /16 CIDR block. There are three private subnets in different Availability Zones. The private subnets each have a large enough CIDR block (/24) to accommodate the nodes in the cluster. We also have three public subnets configured. These public subnets host the NAT, bastion hosts, and OpsCenter, which is the Cassandra utility that monitors a running

Cassandra cluster. The NAT is useful when the cluster nodes in the private subnet need to communicate with the external world for things like software updates. Bastion hosts allow SSH access to your cluster nodes for administrative purposes. To provide access control, our basic setup also associates each layer with a security group.

The architecture diagram shows the seed nodes separately to emphasize the need for having at least one seed per Availability Zone. However, we recommend not to make all the nodes seeds, which can negatively affect your performance due to the use of Gossip. The general rule of thumb is to have at most two seed nodes per Availability Zone.

Cassandra has multiple consistency modes for read and write operations.[26] In a cluster with replication factor of 3, LOCAL_QUORUM operation for read or write allows an operation to succeed when two out of three replicas in a single region signal a success. Because the cluster is spread across three Availability Zones, your local quorum read and write operations continue to be available even during the improbable event of an Availability Zone failure. They also remain available during node failures.

In our basic cluster, OpsCenter nodes are set up in a failover configuration. There are two OpsCenter nodes, which monitor each other on stomp channels.[27] One of them is configured as a primary node. In the event of a failure on the primary, the backup node takes over the role of the primary.

So far, the architecture described is manual. If a node in the Cassandra cluster fails, you have to manually replace it and configure a new node in its place. If the OpsCenter primary node fails, the backup node takes over, but you will need to use the IP address or Domain Name System (DNS) address of the new node. You also need to replace the failed primary node. If the bastion host fails, the replacement has to be manually performed.

Automating This Setup

You can automate most of this setup in AWS. Let us take one layer at a time to show you the AWS components that can be used for this automation.

Let us take the Cassandra cluster first. The steps taken to replace a failed node in Cassandra are pretty static, and manual actions with static steps can be automated. Thus, if you are able to bundle the steps for dead node replacement, you can use Auto Scaling to automate these steps for multiple nodes.[28]

To do so, you can set up an Auto Scaling group with minimum, maximum, and desired size set to the same size.[29] Doing this will allow Auto Scaling to bring up a new node in the place of a failed node when a node fails. The automation that you built can be added to bootstrap the node with the software and induct it into the ring by replacing the failed node. Once this is complete, Cassandra software takes care of bringing the data in the node into the current state by streaming data from other nodes in the cluster. You might need to define separate Auto Scaling groups for seeds and nonseeds. If you require more control over placement, you can choose to create an Auto Scaling group for each Availability Zone and maintain the distribution.

Similarly, you can add the bastion hosts to an Auto Scaling group to allow replacement of failed nodes if there is an Availability Zone or node failure.

For information on how to create a NAT gateway for your cluster, refer to NAT Gateway in the *Amazon VPC User Guide*.[30]

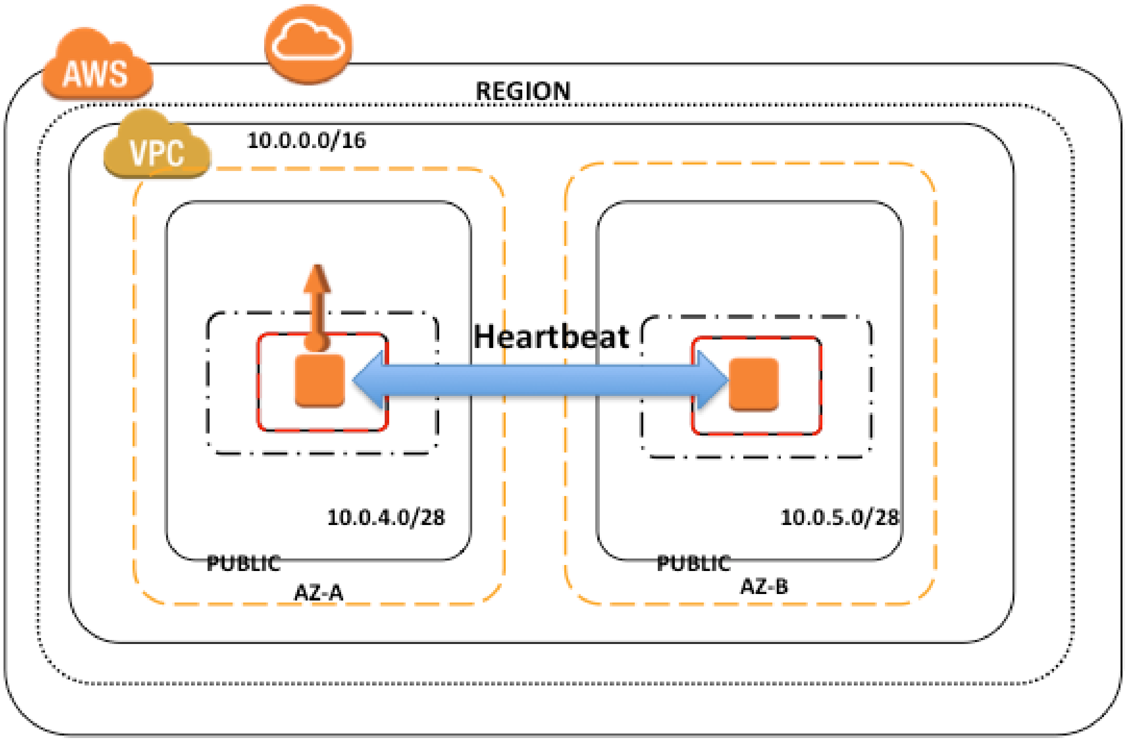

Figure 6: Highly Available OpsCenter

The preceding diagram shows the setup that we recommend for OpsCenter in a high availability configuration. The OpsCenter nodes have a master-slave configuration. To always map the master to one static IP address, you can use an Elastic IP address.[31] For a failover approach, you can deploy the OpsCenter master and slave nodes in individual Auto Scaling groups with a minimum, maximum, and desired size of 1. Doing this will make sure that if Auto Scaling terminates a node in the event of a node failure, Auto Scaling will bring a new node to replace the failed node. The automation can then configure the new node to become the new backup node. During a failover event, the automation should be able to detect the failover and remap the elastic IP to the new master. This approach will allow you to use the static Elastic IP to access OpsCenter for monitoring.

Let us now look at how the cluster architecture looks like with Auto Scaling for Cassandra, shown in the following diagram.

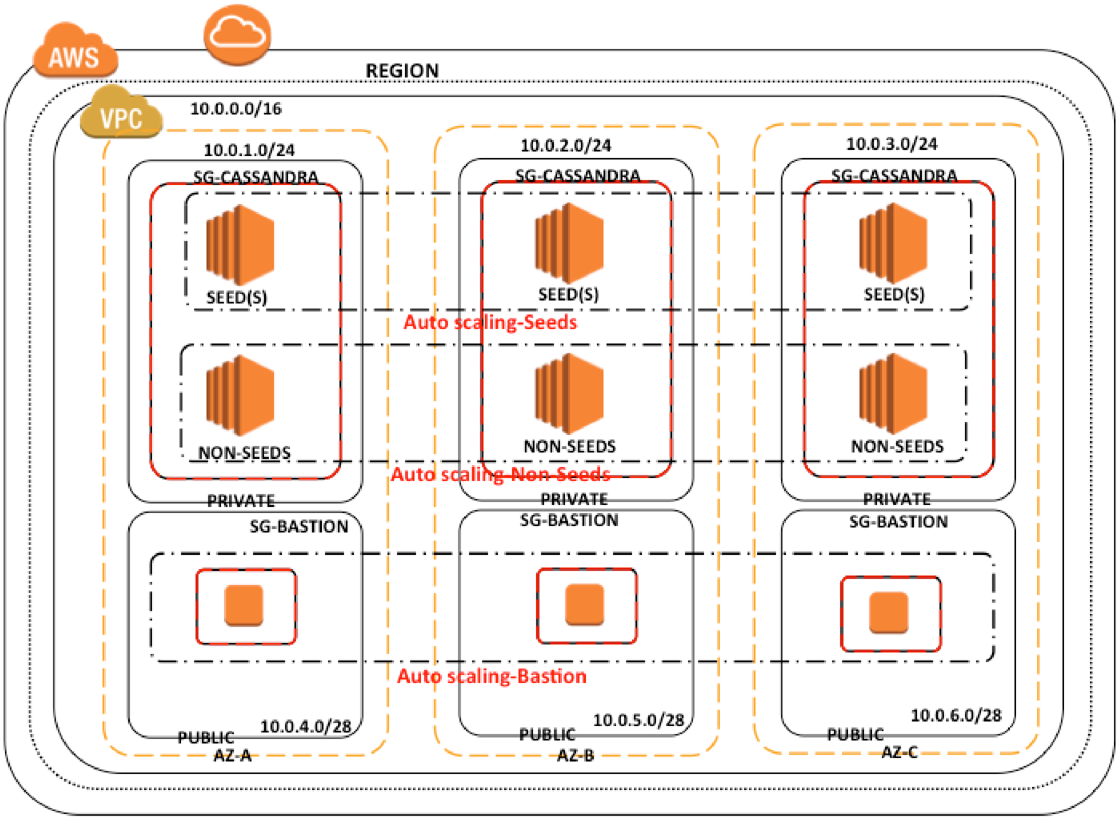

Figure 7: Highly Available Cassandra with Auto Scaling

The best practice is to keep the seed node list consistent across all nodes in the cluster. You might be able to get a list of IP addresses for the seed nodes during the initial launch of the cluster. When a seed node fails and a new node is brought up in its place, the new node is assigned a new IP address. This functionality means that you will need to replace the seed node list across all other nodes in the cluster to reflect the new IP address. Doing this can be a challenge on multiple levels.

You can handle this by creating an Elastic Network Interface (ENI).[32] You can create one ENI per seed node, as described earlier in this whitepaper. Each ENI comes with a static private IP address, and you can add the IP address of the ENI to the seed list. Automation for the seed nodes can then attach the ENI to the new instance spun up by the Auto Scaling group to replace the failed seed instance.

Setting Up for Security

Our high availability setup addresses security concerns. In this setup, nodes in each private subnet have a security group to control inbound network access. However, nodes in the private subnets do not have a route to the Internet. For software updates, when the Cassandra nodes need to talk to the Internet, you can create a NAT instance in a public subnet or configure NAT Gateway that can route the traffic from the Cassandra cluster in the private subnet.

If you have a need to use SSH to access a node for troubleshooting, a bastion host in the public subnet will allow access to any node in the private subnets. For information on how you can configure a bastion host for AWS, see this example on the AWS Security Blog.[33]

In this setup, the OpsCenter hosts are configured with security groups that allow access only from specific instances in your network.

If your compliance requirements require you to have encryption at rest for your Cassandra cluster, you can leverage Amazon EBS volumes with encryption enabled. Amazon EBS encryption uses AWS Key Management Service (AWS KMS) customer master keys for encryption.[34] The encryption occurs on the servers that host EC2 instances, providing encryption of data in transit from EC2 instances to EBS storage. For more details, refer to the EBS encryption documentation.[35]

Encrypting data in transit can be configured following the DataStax documentation.[36]

The following table shows the default set of ports for Cassandra. You can also choose to replace these default ports with nondefault ones. Using these ports will allow you to build a minimal privilege model and further enhance security through the security groups.

Port number	Description	Port Type	Whitelist On
22	SSH port	Public	All nodes
8888	OpsCenter website. The opscenterd daemon listens on this port for HTTP requests coming directly from the browser.	Public	OpsCenter servers
7000	Cassandra internode cluster communication.	Private	Cassandra cluster nodes
7001	Cassandra Secure Socket Layer (SSL) internode cluster communication.	Private	Cassandra cluster nodes
7199	Cassandra JMX monitoring port.	Private	Cassandra cluster nodes
9042	Cassandra client port.	Private	Cassandra cluster nodes
9160	Cassandra client port (Thrift).	Private	Cassandra cluster nodes
61620	OpsCenter monitoring port. The opscenterd daemon listens on this port for Transmission Control Protocol (TCP) traffic coming from the agent.	Private	Cassandra cluster nodes, OpsCenter servers
61621	OpsCenter agent port. The agents listen on this port for SSL traffic initiated by OpsCenter.	Private	Cassandra cluster nodes, OpsCenter servers

Monitoring by Using Amazon CloudWatch

Amazon CloudWatch is a monitoring service for AWS cloud resources and applications you run on AWS. You can use Amazon CloudWatch to collect and track metrics, to collect and monitor log files, and to set alarms. Amazon CloudWatch can send an alarm by Amazon Simple Notification Service (Amazon SNS) or email when user-defined thresholds are reached on individual AWS services. For example, you can set an alarm to warn of excessive CPU utilization.

Alternatively, you can write a custom metric and submit it to Amazon CloudWatch for monitoring. For example, you can write a custom metric to check for current free memory on your instances, and to set alarms or trigger automatic responses when those measures exceed a threshold that you specify. You can also publish JMX or NodeTool metrics to CloudWatch. You can then configure alarms to notify you when the metrics exceed certain defined thresholds.

To publish metrics for Cassandra into CloudWatch, you should use AWS Identity and Access Management (IAM) roles to grant permissions to your instances. You can then use the AWS Command Line Interface (AWS CLI) to publish any metrics directly into CloudWatch. The following example demonstrates how to publish a simple metric named `CompactionsPending` into CloudWatch with a value of 15.

```
aws cloudwatch put-metric-data --metric-name
CompactionsPending --namespace Cassandra --timestamp 2015-
12-13T19:50:00Z --value 15 --unit
```

For further information, see the DataStax documentation on monitoring.[37]

Using Multi-Region Clusters

An example multi-region cluster in Cassandra appears following.

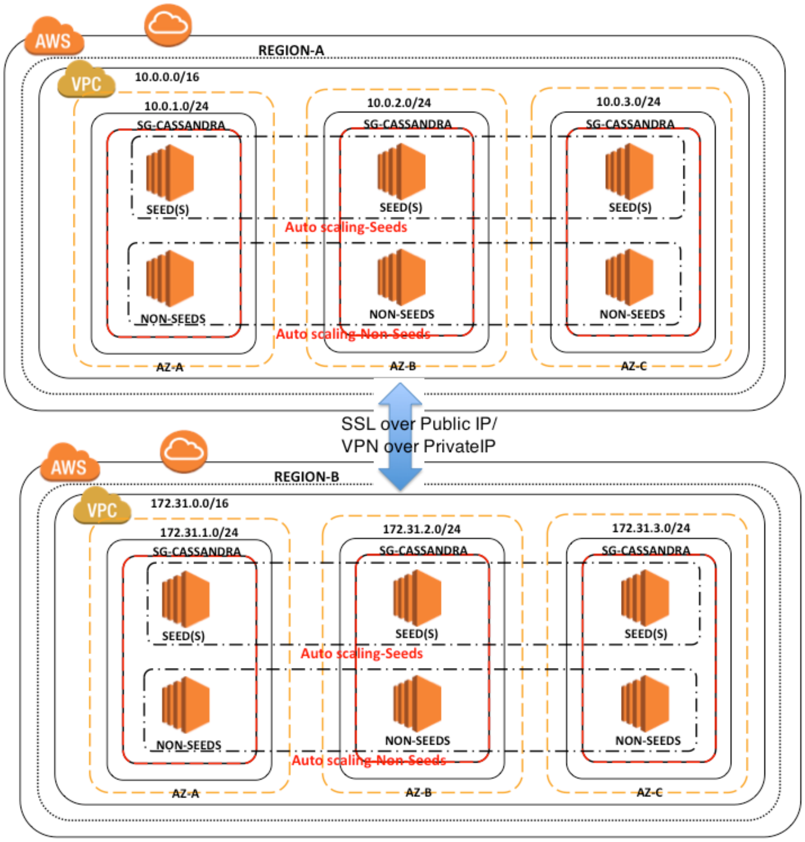

Figure 8: Multi-Region Cassandra Cluster

To set up multi-region clusters, you can enable multi-region cluster communication through Internet Protocol Security (IPsec) tunnels by configuring a virtual private network (VPN). This approach will allow you to use your cluster's private IP address for cross-region communication. You can use either the Ec2Snitch or gossippingpropertyfilesnitch for your snitch when using a VPN for cross-region communication. For more information on setting this up, see the AWS article Connecting Multiple VPCs with EC2 Instances (IPSec).[38]

You can also allow cross-region communication using public IP addresses over the Internet. You should consider using EC2MultiRegionSnitch for this setup. This snitch automatically sets the broadcast address from the public IP address obtained from the EC2 metadata API. You can secure this communication by configuring SSL for your internode communication. For more information, see enabling node-to-node encryption in the DataStax documentation.[39] However, you should set the *listen_address* parameter to use private IP addresses. This approach allows intraregion communication to use private IP addresses instead of public IP addresses.

Another best practice we recommend is to perform read and write operations at a consistency level of LOCAL_QUORUM. If you instead use a global quorum, the read and write operations have to achieve a quorum across AWS regions, which might cause an increase in latency on the client side.

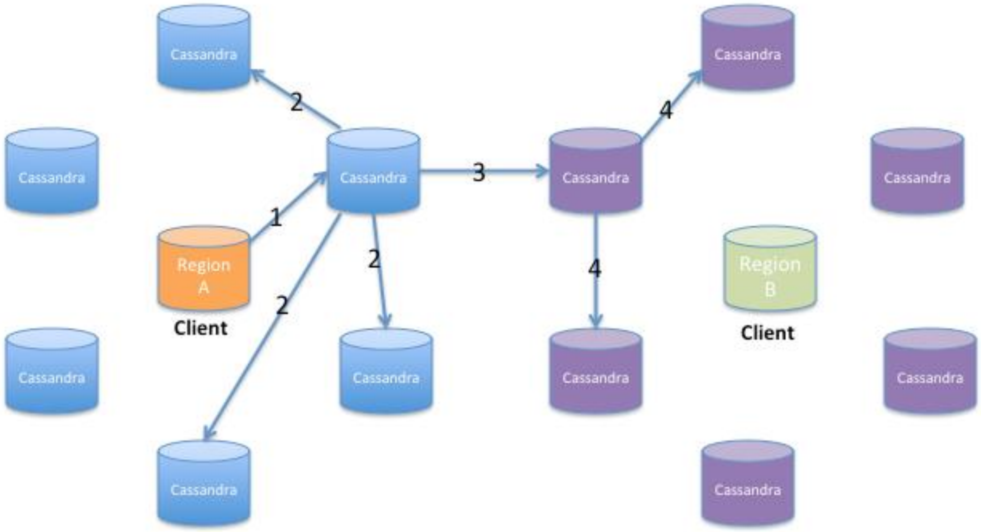

Figure 9: Multi-Region Write Request

The illustration preceding shows how the client performs a write operation using the local quorum in a multi-region write scenario. The following list shows the process:

1. A client sends a write request to a coordinator node in the cluster.
2. The coordinator node sends the request to the appropriate data nodes. At this point, if two out of three nodes in the local region return an acknowledgement, the client's write operation succeeds.
3. The local coordinator forwards the write operation asynchronously to a remote coordinator in the other region.
4. The remote coordinator now sends the data to the remote replicas. The remote replica nodes acknowledge back to the local coordinator.

If a node or region goes offline, hinted handoff can complete a write operation when the node comes back up. You can run nightly repair jobs to keep the regions consistent.[40]

Performing Backups

To take a snapshot of your keyspace for backup purposes, you can use the NodeTool snapshot command. This command will create hard links of your SStables in a snapshots directory inside your keyspace data directory. Note that this command will create a snapshot only on the single node on which the command is run. To snapshot an entire cluster, the command needs to be run on all the nodes in the cluster using parallel SSH or by specifying hostname/IP address to the NodeTool command from a single instance. The SStables can then be backed up to Amazon S3 or Amazon EBS. With EBS, you can snapshot the volume and delete the volume to save on costs.

The preceding method requires significant effort from your side to build, test, and automate a working backup and restore process. If you'd prefer to avoid that effort, consider using the Netflix-built and open-sourced tool Priam.[41] This tool is great for Cassandra backup and recovery. Priam backs up Cassandra to S3 incrementally and provides APIs to restore and recover the cluster from S3.

Another simple approach for backups that works with Amazon S3 is to use tablesnap tool. This tool simply watches for new files in the keyspace directory and uploads those files to S3 from each node.[42]

Building Custom AMIs

Creating your own custom Amazon Machine Image (AMI) for your Cassandra clusters is a good practice. However, you might not want to start from scratch when you are bootstrapping a new Cassandra node.

To create a custom AMI for your Cassandra clusters, you should start with identifying and building your foundation AMI. Here, one of the steps might be to start with a base Linux distribution. You can then build a base AMI on top of the foundation AMI, for example by installing a stable JDK, required agents, and so on. When you have a base AMI, you can run customizations on top of it and build a custom AMI. The custom AMIs that contain the Cassandra software packages installed can be used as a golden image (that is, a template) during your deployment.

As your organization grows bigger, setting up AMIs might help support many of your processes. To learn more about AMI management on AWS, refer to this whitepaper.[43]

Migration into AWS

The following steps migrate an existing Cassandra cluster into EC2 on AWS with zero downtime.

1. Decide on a good starting point for your instance type and storage for this cluster based on the recommendations made earlier in this whitepaper.
2. Test and benchmark your instance for a representative workload either using cassandra-stress or YCSB (described following), Jmeter, or your favorite tool.
3. Once you are satisfied with the instance type and storage, provision a new Cassandra cluster in EC2 in the region of your choice. Note that you might need to change your snitch setting to use either EC2Snitch or EC2MultiRegionSnitch, as described earlier.
4. You should now configure your schema on the new cluster in EC2. This schema will be duplicated from your existing cluster.

5. For a more consistent network experience than the Internet, you might choose to use AWS Direct Connect.[44]

6. Make sure the client is using LOCAL_QUORUM on the existing cluster. This step makes sure that you do not experience elevated latencies because of the addition of a new datacenter.

7. Update the keyspace to replicate data to the EC2 cluster asynchronously. This step will allow the Cassandra cluster on EC2 to get the new write operations.

8. Validate your application logs to make sure that there are no errors and that application latencies are normal.

9. Run the NodeTool rebuild command on the nodes on the EC2 cluster. The rebuild operation can be I/O intensive. If you want this operation to have less impact on the existing cluster, you should consider running the operation on one node at a time. But if your cluster can take additional IO, you can always ramp up and run the operation on multiple nodes at a time.

10. Once the rebuild is completed on all EC2 nodes on the AWS side, you will have all the data synced on the EC2 cluster. Perform validation to make sure the new cluster is working as expected.

11. You can optionally choose to increase the consistency level for write operations to EACH_QUORUM. Doing this will allow the write operations to succeed only if both your current cluster in your datacenter and the EC2 cluster acknowledge they have received the write operation. However, this approach can result in higher latency on your client side.

12. If you enabled EACH_QUORUM, monitor your application logs to make sure that the latencies are as expected, that there are no errors, and that the cluster is responding normally to requests.

13. At this point, you can switch your application to talk to the EC2 cluster.

14. You should update your client to reduce the consistency level back to LOCAL_QUORUM.

15. Again, monitor your application logs to make sure that the latencies are as expected, that there are no errors, and that the cluster is responding normally to requests.

16. If you see any issues, you can use the steps described in this section to perform a rollback.
17. You can optionally migrate your client stack over to AWS to reduce network latencies.
18. You can now decommission the old cluster.

Analytics on Cassandra with Amazon EMR

To help perform analytics on Cassandra, DataStax has released an open source Apache Spark Cassandra driver.[45] Apache Spark is a fast and general processing engine compatible with Hadoop data, and the DataStax driver allows you to use Cassandra column families as Apache Spark Resilient Distributed Datasets (RDDs) . You can use this driver to build Spark applications that can read from and write to Cassandra, allowing you to combine the power of the massively parallel processing in-memory analytic processing capabilities of Spark with Cassandra.

To manage Apache Spark clusters, consider using Amazon Elastic MapReduce (Amazon EMR), a web service that makes it easy to quickly and cost-effectively process vast amounts of data.[46] Apache Spark on Hadoop YARN is natively supported in Amazon EMR, and you can quickly and easily create managed Apache Spark clusters from the AWS Management Console, AWS CLI, or the Amazon EMR API.

You can simply create a cluster with Spark installed on EMR.[47] Now, with Spark readily available on the EMR cluster, you can install the Spark Cassandra driver and its dependencies to create Spark applications that talk to your Cassandra cluster in AWS.

Following is simple example code that you can run on the Spark Shell on the EMR master node to get the count of items in a Cassandra column family named `kv` in a keyspace test.

```
import org.apache.spark.SparkConf
val conf = new
SparkConf(true).set("spark.cassandra.connection.host",
"<Cassandra Listen IP Address>")
```

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
sc.stop
val sc = new SparkContext("local[2]", "test", conf)
import com.datastax.spark.connector._
val rdd = sc.cassandraTable("test", "kv")
println(rdd.count)
```

Amazon EMR also allows you to launch Apache Zeppelin as a sandbox application.[48] Zeppelin is an open source GUI that creates interactive and collaborative notebooks for data exploration. Apache Zeppelin supports an interpreter for Cassandra. You can configure this interpreter in Zeppelin on the Amazon EMR cluster. Doing this will allow you to create interactive notebooks to run Cassandra Query Language (CQL) on Cassandra from Zeppelin to visualize the data in your Cassandra cluster on AWS.

Optimizing Data Transfer Costs

As discussed preceding, we recommend that intraregion communication within the cluster happens with private IP addresses by setting the *listen_address* and *rpc_address* parameters to private IP addresses instead of public IP addresses. However, data transfer charges between EC2 instances always apply when you use public IP addresses for communication. With private IP addresses, charges apply when the communication is between instances in different Availability Zones.

To simplify data transfer and keep costs down, you can perform a number of optimizations. For example, normally when a request is made from a client, the request is sent to any node in the cluster that acts as a coordinator. The coordinator will then send the data to the nodes that own the keyspace. However, the coordinator is an additional step in the request workflow, which can be avoided in most cases by using a client that is token-aware. This approach can potentially avoid this additional hop and increase performance. To find a client driver for this approach, see this list of DataStax supported client drivers.[49]

To further optimize by routing requests to the same Availability Zone as the client, consider Astyanax, a Java client for Cassandra that was open-sourced by Netflix (among several other tools).[50] This client was specifically written with EC2 in mind and allows you to route requests to the same Availability Zone as the client. For example, if your client is in the US East (N. Virginia) region, that is us-east-1a, the client can identify the nodes responsible for the key range and the node that is in the same Availability Zone as the client (us-east-1a in this example) and send the request to that node. This approach can help with performance and optimize data transfer costs by avoiding extra hops.

You can also optionally look at whether internode compression is enabled or not. It is set to all by default, which means that all communication between nodes is compressed. This approach can reduce the amount of network traffic sent across the network pipe and can help with optimizing data transfer costs. However, using this parameter places a low overhead on CPU. You should test to make sure that this parameter is not affecting the performance of your cluster before using it.

Disabling read repair, a property of the table, will further reduce cross-AZ costs. When this property is enabled, some requests will "over-read" from more nodes than required by the specified consistency level. This functionality lets them repair inconsistencies in the background. The speculative retry feature can also result in more data being read, but in this case it might be running because the read operation is taking a long time to complete.

Benchmarking Cassandra

You might want to benchmark Cassandra performance with multiple instance types and configurations before you decide on the right configuration for your cluster. Cassandra-stress is a great tool for benchmarking Cassandra. We highly recommend using this tool to benchmark Cassandra on AWS.[51]

If you want to benchmark Cassandra against other NoSQL engines with a common tool, you can use YCSB.[52] This tool has support for multiple NoSQL engines, including but not limited to Amazon DynamoDB, MongoDB, and Aerospike.

If you want to compare providers through benchmarking, it is generally a good idea to make an apples-to-apples comparison. To make this happen, some of the common things to consider are the following:

1. You should make sure that the Cassandra cluster and the clients from which the benchmarking is run are on the same provider. For example, you might make sure your Cassandra cluster and clients are running in AWS instead of running a Cassandra cluster on AWS and the clients on another provider.
2. You should provision an equivalent amount of resources such as CPU, memory, IOPS, and network. If you cannot achieve equivalence because of the unavailability of comparable instance types between providers, make sure this lack of equivalence is reflected, adjusted for, and accounted for in the results.
3. You should make sure that the configuration parameters for Cassandra, such as the environment and YAML settings, are the same when you are building clusters with multiple providers for benchmarking.

You should ensure that all the optimizations (if any) performed on one platform should be replicated and performed on the other platform to get a fair comparison.

Using the Cassandra Quick Start Deployment

AWS Quick Start reference deployments help you deploy fully functional enterprise software on the AWS cloud, following AWS best practices for security and availability.[53]

We provide a Cassandra Quick Start that automatically launches and runs a Cassandra cluster on AWS. It automates the deployment through an AWS CloudFormation template, and enables you to launch the Cassandra cluster either into your own Amazon VPC or into a newly created Amazon VPC. Customization options include the Cassandra version you want to deploy (version 2.0 or 2.1), the number of seeds and nonseeds you want to launch to ensure high availability (one to three seeds), automatic node replacement with Auto Scaling, OpsCenter for monitoring, NAT gateway integration with the VPC configuration, and a choice of Java versions (Java 7 or Java 8).

The Cassandra Quick Start takes approximately 15 minutes to deploy. You pay only for the AWS compute and storage resources you use—there is no additional cost for running the Quick Start. The templates are available for you to use today. The CloudFormation template to build the cluster in a new VPC is available here, [54]and the CloudFormation template to build the cluster in an existing VPC is available here.[55]

Conclusion

The AWS cloud provides a unique platform for running NoSQL applications, including Cassandra. With capacities that can meet dynamic needs, costs based on use, and easy integration with other AWS products, such as Amazon CloudWatch, AWS Cloud Formation, and Amazon EBS, the AWS cloud enables you to run a variety of NoSQL applications without having to manage the hardware yourself. Cassandra, in combination with AWS, provides a robust platform for developing scalable, high-performance applications.

Contributors

The following individuals and organizations contributed to this document:

- Babu Elumalai, Solutions Architect, Amazon Web Services

Further Reading

For additional help, please consult the following sources:

- Regions and Availability Zones
- Amazon VPC Documentation
- Amazon EBS
- Amazon EC2 FAQ

- AWS Security Center
- DataStax Cassandra 2.1 Documentation

Notes

[1] https://aws.amazon.com/dynamodb/

[2] https://aws.amazon.com/s3/

[3] http://docs.datastax.com/en/landing_page/doc/landing_page/current.html

[4] https://en.wikipedia.org/wiki/Log-structured_merge-tree

[5] https://en.wikipedia.org/wiki/Quorum_(distributed_computing)

[6] https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_how_cache_works_c.html

[7] https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/dml_about_hh_c.html

[8] http://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsRepair.html

[9] https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsTuneJVM.html

[10] https://aws.amazon.com/about-aws/global-infrastructure/

[11] https://aws.amazon.com/about-aws/global-infrastructure/

[12] https://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architectureSnitchesAbout_c.html

[13] https://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architectureSnitchEC2_t.html

[14] https://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architectureSnitchEC2MultiRegion_c.html

[15] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html#enhanced_networking_instance_types

[16] http://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_keyspace_r.html

[17] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/vpc-classiclink.html

[18] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-eni.html#AvailableIpPerENI

[19] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html#EBSVolumeTypes_gp2

[20] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html#EBSVolumeTypes_piops

[21] http://docs.datastax.com/en/cassandra/2.1/cassandra/planning/architecturePlanningEC2_c.html

[22] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSOptimized.html

[23] https://aws.amazon.com/ec2/pricing/

[24] https://www.youtube.com/watch?v=1R-mgOcOSd4

[25] https://aws.amazon.com/ec2/pricing/

[26] http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html

[27] http://stomp.github.com/

[28] http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/WhatIsAutoScaling.html

[29] http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/AutoScalingGroup.html

[30] https://aws.amazon.com/articles/2781451301784570

[31] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

[32] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-eni.html

[33] https://blogs.aws.amazon.com/security/post/Tx3N8GFK85UN1G6/Securely-connect-to-Linux-instances-running-in-a-private-Amazon-VPC

[34] https://aws.amazon.com/kms/

[35] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSEncryption.html

[36] https://docs.datastax.com/en/cassandra/2.1/cassandra/security/secureSslEncryptionTOC.html

[37] https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_monitoring_c.html

[38] http://aws.amazon.com/articles/5472675506466066

[39] http://docs.datastax.com/en/cassandra/2.1/cassandra/security/secureSSLNodeToNode_t.html

[40] https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/dml_about_hh_c.html

[41] https://github.com/Netflix/Priam

[42] https://pypi.python.org/pypi/tablesnap

[43] https://d0.awsstatic.com/whitepapers/managing-your-aws-infrastructure-at-scale.pdf

[44] https://aws.amazon.com/directconnect/

[45] https://github.com/datastax/spark-cassandra-connector

[46] https://aws.amazon.com/elasticmapreduce/

[47] https://docs.aws.amazon.com/ElasticMapReduce/latest/ReleaseGuide/emr-spark-launch.html

[48] http://docs.aws.amazon.com/ElasticMapReduce/latest/ReleaseGuide/emr-sandbox.html

[49] http://www.planetcassandra.org/client-drivers-tools/#%20DataStax%20Supported%20Drivers

[50] https://github.com/Netflix/astyanax/wiki/Getting-Started

[51] http://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsCStress_t.html

[52] https://github.com/brianfrankcooper/YCSB

[53] http://aws.amazon.com/quickstart/

[54] https://s3.amazonaws.com/quickstart-reference/cassandra/latest/templates/Cassandra_EBS_VPC.json

[55] https://s3.amazonaws.com/quickstart-reference/cassandra/latest/templates/Cassandra_EBS_NoVPC.json