Running Neo4j Graph Databases on AWS

May 2017

Notices

Contents

Abstract

Amazon Web Services (AWS) is a flexible, cost-effective, and easy-to-use cloud computing platform. Neo4j is the leading NoSQL graph database that is widely deployed in the AWS Cloud. Running your own Neo4j deployment on Amazon Elastic Compute Cloud (Amazon EC2) is a great solution for users whose applications require high-performance operations on large datasets.

This whitepaper provides an overview of Neo4j and its implementation on the AWS Cloud. It also discusses best practices and implementation characteristics such as performance, durability, cost optimization, and security.

Introduction

NoSQL refers to a subset of structured storage software that is optimized for high-performance operations on large datasets. As the name implies, querying of these systems is not based on the SQL language—instead, each product provides its own interface for accessing the system and its features.

A common way to understand the spectrum of NoSQL databases is by looking at their underlying data models:

- Column stores – Data is organized into columns and column families providing a nested hashmap-like structure accessed by key.
- Key-value stores – Data is organized as key-value relationships and accessed by primary key.
- Document databases – Data is organized as documents (e.g., JSON, XML) and accessed by fields within the document.

Neo4j provides a far richer data model than other NoSQL databases. Instead of working with isolated values, columns, or documents, Neo4j supports relationships between data so that webs of interconnected data can be created and queried. We see this kind of data every day in use cases from social networks to transport, road, and rail networks. Graph databases are already widely applied in fields as diverse as healthcare, finance, education, IT infrastructure, identity management, Internet of Things (IoT), and many more.

In this whitepaper, we'll discuss how to run Neo4j effectively on AWS. The on-demand nature of Amazon Elastic Compute Cloud (Amazon EC2) and the power of Neo4j together provide a great way for you to deploy graph data to support your use case while avoiding the undifferentiated heavy lifting typically associated with purchasing, deploying, and managing traditional infrastructure.

Transacting with the Graph

To make traversing the graph efficient and safe from physical and semantic corruption, the graph model demands strong consistency with its underlying storage. That is, if a relationship exists between two nodes, it must be reachable from both of them.

Graph Consistency
If the records in a graph database disagree about connectivity, a non-deterministic structure will result. Traversing the graph in one direction leads to different actions being taken than if the graph were traversed in the other direction. This, in turn, leads to different decisions being recorded in the graph, which may lead to semantic corruption spreading throughout the graph, compounding the initial physical corruption.

In order to preserve the rigorous consistency required for graphs, Neo4j uses atomic, consistent, isolated, and durable (ACID) transactions when modifying the graph. In the case of read-only transactions, the cost of transactions is minimal because read locks do not block other reads, and there is no need to flush to disk. To ensure safe, recoverable write transactions, the system will take write locks which will block reads and flush data to the transaction log before completing the transaction.

To reduce the impact of a physical flush, Neo4j amortizes the cost of flushing across multiple small concurrent transactions. This means thousands of ACID transactions per second can be processed in a well-tuned system while preserving safety.

With Amazon EC2 there are multiple instance types that feature high-performance solid-state drives (SSDs) that vastly reduce the cost of writing to disk. Therefore it’s possible to tune for a greater number of transactions per second based on high-performance block storage. Such performance is available in the I2 instance family, which is designed to perform up to 300,000 input/output operations per second (IOPS).

Deployment Patterns for Neo4j on AWS

Neo4j differs from other database management system (DBMS) engines in that it can either be deployed as a traditional database server or embedded within an application. This bimodal operation provides the same APIs, the same transactional guarantees, and the same level of cluster support either way. In the following sections, we describe the deployment model of a traditional database server that is deployed on Amazon EC2.

Basics

Neo4j was originally conceived as an embedded Java library intended to provide idiomatic access to connected data through a graph API. While Neo4j retains the ability to be embedded in JVM-based applications, it has grown in sophistication since those days, adding an excellent query language, practical programmatic APIs, and support for high availability (HA) via clusters of Neo4j instances. That functionality can be invoked over the network from any platform despite the 4j naming!

Neo4j is a transactional database that supports high concurrency while ensuring that concurrent transactions do not interfere with each other. Even deadlocking transactions are automatically detected and rolled back.

When data is written to Neo4j, it's guaranteed durable on disk. In the event of a fault, no partially written records will exist after restart and recovery. A single instance of the database is resilient right up to the point where the disk is lost.

To protect against the failure of a disk, Neo4j has an HA mode in which multiple instances of Neo4j can collaborate to store and query the same graph data. The loss of any individual Neo4j instance can be tolerated since others will remain available. In fact, work proceeds as usual when a majority of the Neo4j cluster is available. Neo4j is able to capitalize on the robust features that AWS offers not only to detect failures, but also to provide automated recovery mechanisms.

Networking

Neo4j HA trusts the network and so it's important to physically secure it against intrusion and tampering. Conversely, for application-database interactions, Neo4j supports transport level security (TLS) out of the box for privacy and integrity.

AWS offers a high-performance networking environment in a customer-controlled VPC created with Amazon Virtual Private Cloud (VPC). Within your VPC, you can create and manage the logical network components that you need to deploy your application infrastructure. The VPC enables you to create your own network address space, subnets, firewall rules, route tables, as well as extend connectivity to your own data centers and the Internet.

The network design for a Neo4j cluster can be easily customized to the specific application on AWS. Most customers choose to keep the database on a private subnet that has strict network controls in place to prevent unauthorized network access.

There are two different types of firewalls built into the AWS Cloud that provide a high level of network isolation. The first type is a security group, which is a stateful firewall that is applied at the instance level in both the inbound and outbound directions. The security group defines which protocols, ports, and Classless Inter-Domain Routing (CIDR) IP address ranges have access to a specific instance. Security groups have an implicit deny, which means that there is no network access by default. To be granted network access, a security group must specifically allow the traffic through.

Deploying your Neo4j cluster into a VPC with a private subnet and configuring your security group to permit ingress over the appropriate TCP ports builds another layer of network security. The following table shows default TCP port numbers for Neo4j:

Port	Process
7474	The Neo4j REST API and web frontend are available at this port.
7687	The binary protocol endpoint. Used by application drivers to query and transact with the database.

The second type of firewall is a Network Access Control List (NACL). A NACL is defined at the subnet level and is a stateless firewall. A NACL is an ordered set of rules that is evaluated with the lowest number rule first. By default the NACL has an explicit rule to allow all traffic to flow in both directions on the subnet. However, NACL rules can also be applied to allow traffic to flow in either the inbound or outbound directions as well.

Every Amazon EC2 instance is allocated bandwidth that corresponds to its size, which currently in the X1 instance family can be up to 20 Gbps of network bandwidth. As instance size decreases, so does the bandwidth allocated to the instance. If your application requires a high level of network communication between hosts, ensure that the instance size selected will deliver the bandwidth needed. In Neo4j this typically corresponds to systems that sustain high write loads.

VPC Design

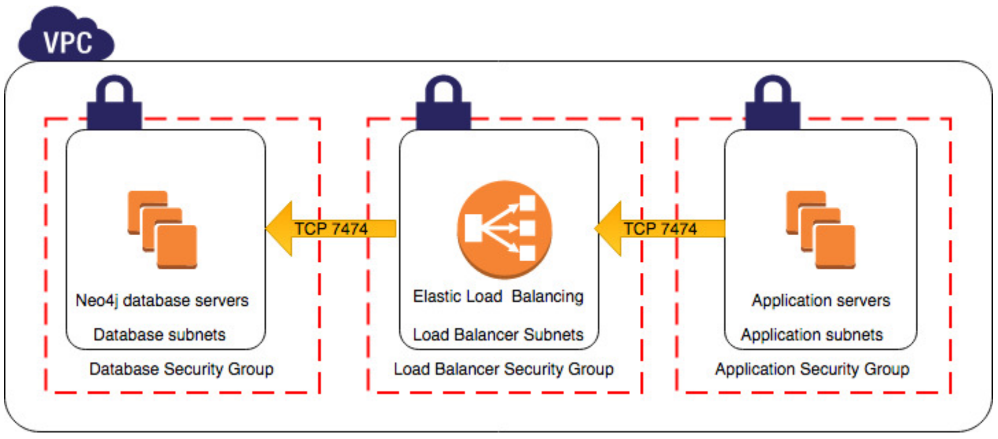

Figure 1: Sample VPC design

To optimize network performance, we suggest using EC2 instances that support enhanced networking, which uses single root I/O virtualization (SR-IOV) to ensure that your instances can achieve greater packets per second, reduced latency, and reduced jitter.

AWS recommends that you use a multi-Availability Zone (AZ)[1] design for your applications in order to achieve a high level of fault tolerance. By using multiple Availability Zones, you can mitigate the risk of an entire Availability Zone failing by replicating to another instance in a separate Availability Zone. With Neo4j, the network latency between instances will increase because the Availability Zones are in separate physical locations.

Clustering

Neo4j HA is available both to server-based and embedded instances of Neo4j as part of Neo4j Enterprise Edition.[2] The clustering architecture has been designed with two features in mind:

- Optimized for graph workloads.
- Simple to understand and operate.

A beneficial side effect of the high availability of Neo4j Enterprise Edition is that it can scale horizontally for graph operations while scaling vertically for transaction processing. This topology is favorable for graph workloads since graphs are intrinsically read-heavy (even when writing to a graph, it must first be traversed to find the right part of the structure to update).

On that basis, Neo4j has opted for a clustering system similar to that found in mature relational databases, in which the cluster members can have either a master or slave role.

A Neo4j HA cluster operates cooperatively because each database instance contains the logic it needs to coordinate with the other members of the cluster. On startup, a Neo4j HA database instance tries to connect to an existing cluster specified by configuration. If the cluster exists, the instance joins it as a slave. Otherwise, the cluster will be created and the instance will become the current master.

Note that the master role is transitory. A master is elected via an instance of the Paxos algorithm embedded in Neo4j. Any machine can instigate an election if it thinks it has detected a fault, but a majority of the machines in the cluster must participate in the election. After the election is complete, one master remains or becomes elected, and all other machines in the cluster become slaves.

Whenever a Neo4j instance becomes unavailable, the other database instances in the cluster detect that and mark it as temporarily failed. A database instance that becomes available after being unavailable will automatically catch up with the latest cluster updates.

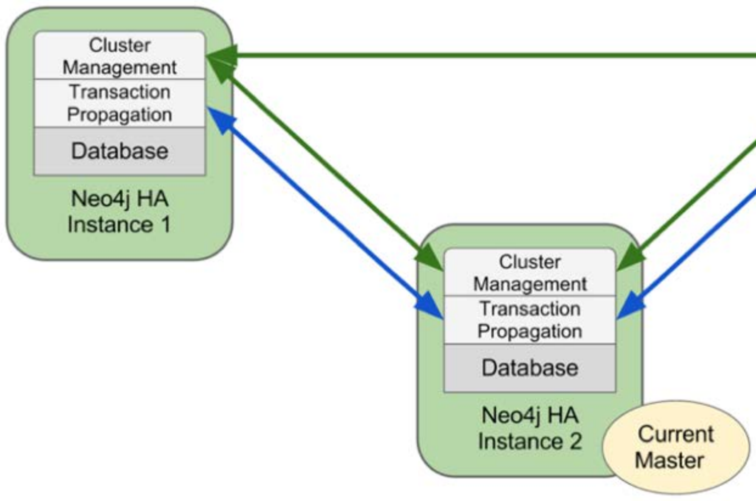

Figure 2: Neo4j HA clusters showing current master

If the master fails, another (best-suited) member will be elected and have its role switched from slave to master after a quorum has been reached within the cluster. When the role switch has been performed, the new master will broadcast its availability to all the other cluster members. A new master is typically elected and started within just a few seconds; during this time no writes can take place.

Be aware that during the transition period, if an old master had changes that did not get replicated to any other member before becoming unavailable and if a new master is elected and performs changes before the old master recovers, there will be two "branches" of the database. The old master will move away its database (its "branch") and download a full copy from the new master to become available as a slave in the cluster. An operator can then choose to replay the transactions in the branched data to the cluster.

Neo4j High Availability

In the Neo4j HA architecture, the cluster is typically fronted by load balancers provided by Elastic Load Balancing or HAProxy. Elastic Load Balancing (ELB) is an AWS service that offers load balancers that automatically distribute traffic across multiple EC2 instances and across multiple Availability Zones. An ELB load balancer is elastic because it automatically scales its request-handling capacity to support network traffic and doesn’t cap the number of connections that it can establish with EC2 instances. If an instance fails, the load balancer automatically reroutes the traffic to the remaining EC2 instances that are running. If the failed EC2 instance is restored, the load balancer restores the traffic to that instance.

Data Integrity
Integrity is crucial for a transactional database. The master role imposes a total ordering of transactions on the system. This has the beneficial side effect that all replicas in the cluster apply transactions in exactly the same order and therefore are kept identical.

Elastic Load Balancing controls and distributes traffic to your EC2 instances and serves as a first line of defense to mitigate network attacks. You can offload the work of encryption and decryption to your load balancer so that your EC2 instances can focus on their main work. Elastic Load Balancing has configurable health checks that can be used in conjunction with Amazon CloudWatch to send alerts and take action when specified thresholds are reached. If Auto Scaling is used with Elastic Load Balancing, instances that are launched by Auto Scaling are automatically registered with the load balancer, and instances that are terminated by Auto Scaling are automatically de-registered from the load balancer.

An ELB load balancer can be Internet-facing or internal, and can accept HTTP, HTTPS, SSL, and TCP connections with the ability to terminate SSL to offload the burden on the backend EC2 instances. Elastic Load Balancing can also bring an extra level of security to your network design because the security groups applied to the Neo4j servers can be configured to only accept traffic from the load balancer, which might help prevent unauthorized access to the instances.

To maintain static connection points to the master database server and to the read replicas from the application, it is suggested that you use two separate load balancers for this. By doing this, your application will not need to be updated when a new master database is elected, a new slave is added, or a failure on one of the nodes occurs.

Neo4j advertises separate REST endpoints for both the master node and the slave nodes so that the load balancers can determine what role each instance in a cluster plays. By creating two load balancers and adding all of the Neo4j instances to both load balancers, we can ensure that during an election the master node load balancer will properly redirect requests to the proper nodes.

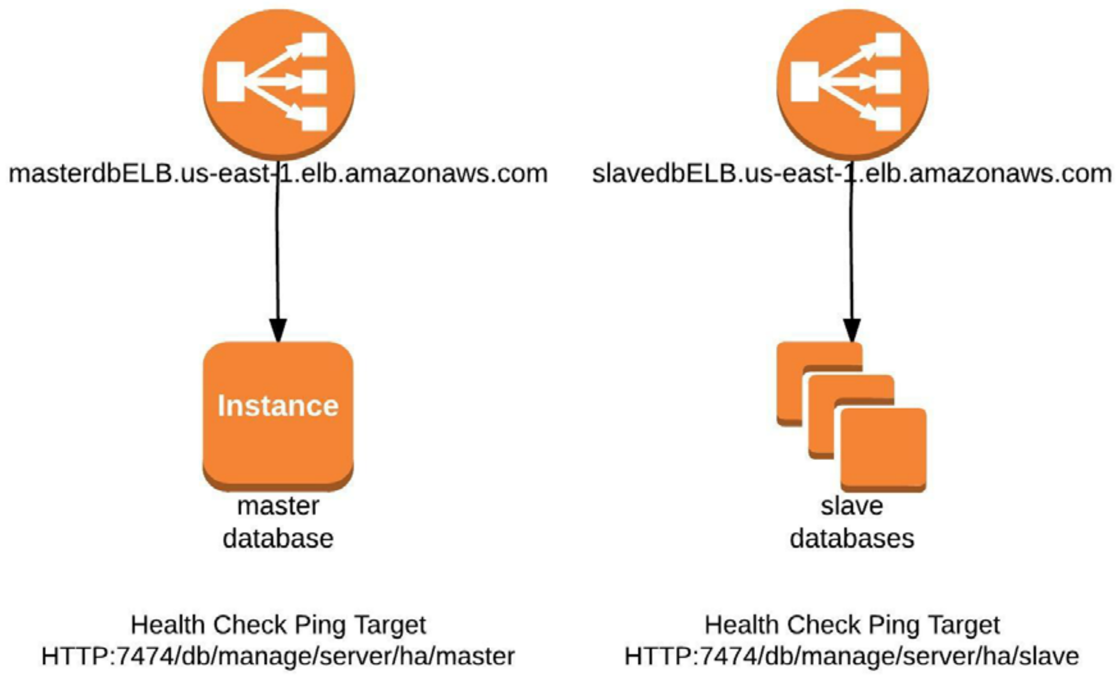
masterdbELB.us-east-1.elb.amazonaws.com
slavedbELB.us-east-1.elb.amazonaws.com
Instance
master database
slave databases
Health Check Ping Target HTTP:7474/db/manage/server/ha/master
Health Check Ping Target HTTP:7474/db/manage/server/ha/slave

Figure 3: Neo4j cluster REST endpoints for the master node and the slave nodes

Master Node Elastic Load Balancer

The master node will respond with a 200 status code with a body text of “true” and the slaves will return a `404 Not Found` with a body text of “false” when the load balancer health check references /db/manage/server/ha/master.

```
"HealthCheck": {
                "HealthyThreshold": 2,
                "Interval": 10,
                "Target":
"HTTP:7474/db/manage/server/ha/master",
                "Timeout": 5,
                "UnhealthyThreshold": 2
            },
```

Slave Node Elastic Load Balancer

The master node will respond with a 404 status code with a body text of “false” and the slaves will return a 200 status code with a body text of “true” when the load balancer health check references /db/manage/server/ha/slave.

```
"HealthCheck": {
                "HealthyThreshold": 2,
                "Interval": 10,
                "Target": "HTTP:7474/db/manage/server/ha/slave",
                "Timeout": 5,
                "UnhealthyThreshold": 2
            },
```

This functionality allows both the master and slave load balancers to respond to Neo4j cluster events without any application changes or administrator involvement.

Database Storage Considerations

AWS provides two fundamental kinds of storage: Amazon Elastic Block Store (EBS) and EC2 ephemeral instance store.

Several EC2 instance types expose multiple ephemeral instance stores that can be used for mirroring data for fault tolerance. However, if the instance stops, fails, or is terminated all data will be lost, and so strategies need to be in place to address those risks. High-speed ephemeral storage is beneficial to graph databases that are larger than the physical memory limit of a single EC2 instance. In the case that the database is larger than the main memory, specific considerations need to be taken since it will not be possible for a single Neo4j instance to cache the whole database in RAM. This means that the portions of the graph that are not frequently accessed will have to run out of memory and on a storage device. The preference would be to maintain extremely rapid in-memory traversals of the graph for the whole graph, no matter its size.

- Amazon EC2 X1 Instances – X1 instances have the lowest price per GB of RAM and are ideally suited for in-memory databases. With up to 1,952 GB of DDR-based memory, 128 vCPUs, and 3,840 GB of SSD storage, the X1 instance is the most performant for the largest Neo4j use cases.
- Amazon EC2 I2 Instances – High I/O (I2) instances are optimized to deliver more than 300,000 low-latency IOPS to applications by utilizing up to 8 SSD drives to minimize access time with a capacity of up to 6,400 GB.

- Amazon EC2 D2 Instances – Dense-storage (D2) instances provide an array of up to 24 internal drives with a capacity of 2 TB each. These disks can be configured with multiple RAID types and partition sizes as needed. A D2 instance can provide up to 3.5 Gbps read and 3.1 Gbps write disk throughput with a 2 MB block size and a capacity of 48 TB.

Neo4j is a shared-nothing architecture and can therefore happily consume instance-based storage. Inevitable data loss when instances are stopped or terminated can be prevented by clustering. We mostly focus on instance storage here, but other uses exist for Neo4j on Amazon EBS that we explore at the end of this section.

EC2 instances can also use Amazon EBS, which provides persistent block-level storage volumes. Amazon EBS volumes are highly available and reliable storage volumes that can be attached to any running instance that is in the same Availability Zone. EBS volumes that are attached to an EC2 instance are exposed as storage volumes that persist independently from the life of the instance. Besides being persistent data stores, you can create point-in-time snapshots of EBS volumes, which are persisted to Amazon Simple Storage Service (S3). Snapshots protect data for long-term durability, and they can be used as the starting point for new EBS volumes. The same snapshot can be used to instantiate as many volumes as you want. These snapshots can be copied across AWS Regions. In a large database, bringing up a cluster from scratch can take time to transfer all of the data between existing and new Neo4j instances. Amazon EBS provides the ability to mount the data store files from a snapshot of another instance and then recover the Neo4j instance atop that store file before it rejoins the cluster. This reduces the overall time that it takes to bring a new Neo4j instance into the cluster.

Storage Scaling

Now that you have learned the fundamentals of clustering Neo4j, let’s look at how the platform can be used to scale out the database. For scaling Neo4j, you need to consider the performance of the database under load and the physical volume of the graph being stored. These two concerns are almost, but not entirely, orthogonal. There are subtle interplays between operational load and low latency data access as you scale both.

Scaling for Volume

Let’s start with understanding scaling for volume, since scaling for performance arises naturally from there. In the Neo4j world, large datasets are those that are substantially larger than main memory. This presents an interesting challenge for performance engineering since RAM provides the best balance of performance and size for database operations (that is, CPU cache is tiny but faster, and disks are larger but slower). Databases love RAM, and Neo4j is no exception to that rule—the more RAM available to the database, the lower the possibility that it runs at disk speed rather than at memory speed. The majority of this memory can be used by the database, in particular consumed by Neo4j's page cache.

Data Consistency
Neo4j is an ACID transactional database. Any abrupt shutdown of an instance, such as when an EC2 instance unexpectedly dies, will leave the database files in an inconsistent but repairable state. Hence, when booting the new Neo4j instance using files on the existing EBS volume, the database will first have to recover to a consistent state before joining the cluster. This process may be much quicker than a full sync from scratch.

Although scaling vertically is always an option thanks to rapid growth in affordable large-memory machines in the AWS ecosystem and the ease of switching from one instance type to another, scaling horizontally offers its own advantages.

Neo4j uses an HA cluster with a pattern called "Cache Sharding" to maintain high performance traversals with a dataset that substantially exceeds main memory space. Cache sharding isn’t sharding in the traditional sense, since we expect a full data set to be present on each database instance for impeccable fault tolerance and to maintain excellent performance when the memory-to-disk ratio is lopsided. But cache sharding allows Neo4j to aggregate the RAM of individual instances by consistently routing like queries to the same database endpoint.

In the typical case where a server supports multiple concurrent clients, access patterns tend to be noisy at first glance, approximating all of the random walks of the graph overall. Yet even at large scale randomness is never truly dominant.

After all, since the graph is structured it makes sense that queries would be structured, too.

With multiple concurrent clients, it’s possible to discern commonality between them. Whether by geography, username, or other application-specific feature, it’s almost always possible to discern a coarse feature of the access pattern on the wire so that like requests can be consistently routed to the same server instance.

The solution architecture for this setup is shown in Figure 4. The technique of consistent routing has been implemented by high-volume web properties for a long time, and it is simple to implement, scales well, and is very robust.

The strategy we use to implement consistent routing will typically vary by domain. Sometimes it’s fine just to use session affinity (commonly called “sticky sessions”) implemented by the Elastic Load Balancing. At other times we’ll want to route based on the characteristics of the data set. A simple strategy is that the instance that first serves requests for a particular user will serve subsequent requests. By doing this, there is a greater chance that a warm cache will process the requests. Other domain-specific approaches will also work. For example, in a geographical data system we can route requests about particular locations to specific database instances that will be warm for that location.

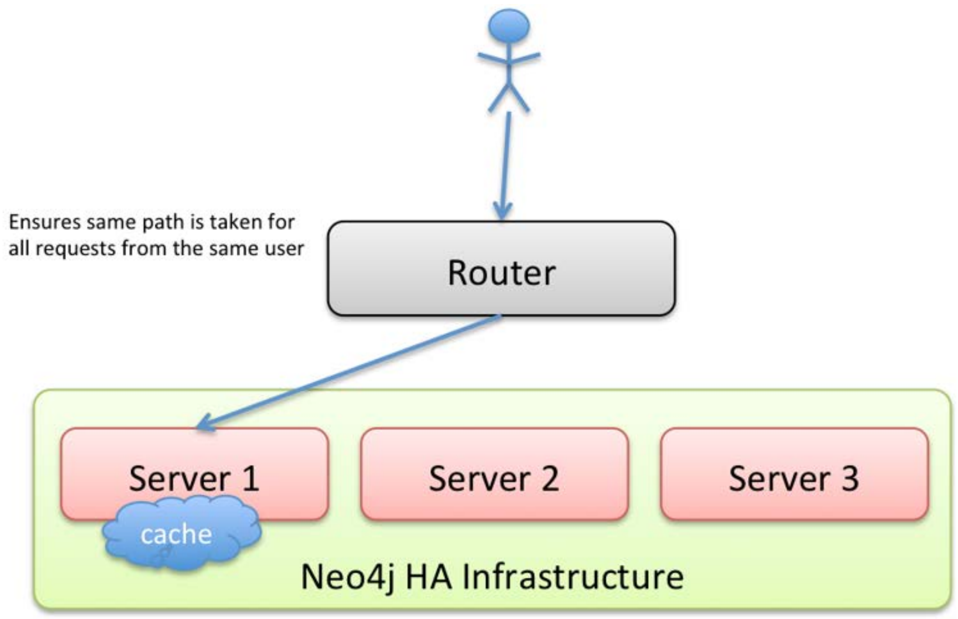
Ensures same path is taken for all requests from the same user
Router
Server 1
Server 2
Server 3
cache
Neo4j HA Infrastructure

Figure 4: Solution architecture for consistent routing

Either way, we're increasing the likelihood of the required graph data already being cached in RAM, which makes traversals extremely performant. Adding high performance block storage to the mix means that even where the cache is cold (e.g., when a machine is restarted or a new part of the graph is being traversed) the cache miss penalty is minimized.

Scaling for Performance

Now that you have seen how Neo4j scales for volume, scaling for performance is simplified by adding more instances. Provided that you can identify suitable workloads that do not decrease cache performance, you can simply add more instances to the Neo4j cluster to support more graph operations at an approximately linear rate.

Cache Sharding
Assuming a uniform distribution, usernames work well with this scheme, and sticky sessions with round-robin load balancing work in almost all domains.

In practice, to get the best performance, your choice of routing key for cache sharding must be able to become finer-grained as the number of servers grows. For example, you could change the routing based on the names of countries beginning *A-G*, *H-N*, and then *O-Z* ultimately to a separate Neo4j database instance for each group. By designing the database this way, it's possible to gain more throughput by adding more machines and using each of those machines as efficiently as possible.

Operations

Operating a Neo4j cluster at scale is similar to running other database servers on Amazon EC2. Like all databases, Neo4j uses working files such as logs as it executes. Logs can be kept for troubleshooting purposes. However, we recommend limiting them in size or temporal scope.

The transaction log is important because this is where Neo4j transactions are made durable before being applied to the data model. This log particularly important in the backup process.

Although you can keep the logical logs forever (and, therefore, rebuild your database from scratch merely by replaying all the transactions in those log files), in practice this would require a lot of storage for a database that has run in production for a reasonable amount of time. In practice, logs are maintained on a schedule that is suitable for troubleshooting and that takes the incremental backup schedule into consideration. (We discuss incremental backups in more detail later.)

Monitoring

AWS offers a monitoring service called Amazon CloudWatch, which provides a reliable, scalable, and flexible monitoring solution for EC2 instances and AWS services. CloudWatch enables near real-time monitoring on multiple EC2 metrics, as well as the ability to monitor customer-supplied metrics. With CloudWatch, alarms and notifications can be triggered based on events, which can quickly alert you to issues, and can apply automation to resolve the issues. Additionally, Amazon CloudWatch Logs provides the ability to collect, store, monitor, and troubleshoot application-level issues. CloudWatch Logs can greatly simplify aggregating the system and application logs from all of the nodes in the Neo4j cluster. CloudWatch Logs is agent-based and enables every EC2 instance in the cluster to perform comprehensive logging.

Configuring CloudWatch with Neo4j

CloudWatch can be configured on an existing EC2 instance[3] or on a new EC2 instance.[4] Once installed, the /etc/awslogs/awslogs.conf file is configured to monitor the Neo4j .log. At the bottom of the awslogs.conf file, the following section will be added:

```
[Neo4j .log]
datetime_format = %Y-%m-%d %H:%M:%S%f%z
file = /home/ec2-user/Neo4j 3/Neo4j -enterprise-3.0.0-
RC1/logs/Neo4j .log
log_stream_name = {instance_id}
initial_position = start_of_file
log_group_name = /Neo4j /logs
```

After the awslogs service is started, check the /var/log/awslogs.log for any errors.

Configuring metrics and alerts for Neo4j is addressed in this Neo4j knowledge base article[5].

Online Backup

Neo4j can be backed up while it continues to serve user traffic (called "online" backup). Neo4j offers two backup options: full or incremental. These strategies can be combined to provide the best mix of safety and efficiency. Depending on the risk profile of the system, a typical strategy might be to have daily full backups and hourly incremental backups, or weekly full backups with daily incremental backups.

As the name suggests, a full backup will clone an entire database. These are the characteristics of a full backup:

- Copies database store files.
- Does *not* take locks.
- Replays transactions run after backup started until end of store file copy.

At the end of a full backup, there is a consistent database image on disk. This backup file can be safely stored away, and recovering to this backup is as simple as copying the database files back into the Neo4j data directory (typically `<Neo4j home>/data/graph.db`).

After the backup has been created, the recommendation is for the backup to be copied from the EC2 instance that ran the process into stable, long-term storage. Amazon S3 provides a range of suitable archive storage platforms depending on your needs. The backup can be copied to Amazon S3 directly, or you can achieve the same level of durability by using an EBS snapshot, which is stored in Amazon S3 automatically.

Amazon EBS is a network-shared storage service that can be mounted from any EC2 instance. Amazon EBS provides persistent block-level storage volumes that are automatically replicated within their Availability Zones to protect from component failure, offering high availability and durability. A snapshot can be created from an EBS volume, which not only provides the ability to restore data in the future, but also provides the ability to mount that volume to another EC2 instance. This process can greatly decrease the time that it takes to add an additional Neo4j node to the cluster. A side benefit of EBS snapshots is that they are persisted to Amazon S3, which means that they are protected for long-term durability. Volumes can be created from snapshots in any Availability Zone in the Region, and snapshots can also be copied across Regions to provide an even greater level of durability.

Amazon S3 provides three tiers of storage optimized for cost versus frequency of access. Amazon also provides lifecycle policies that can automatically transition objects from Amazon S3 Standard to Amazon S3 Infrequent Access and AWS Glacier (for long-term archive) after a specific amount of time has elapsed. Lifecycle policies streamline the archival and cost-saving process so that you don't have to manually transition objects or pay increased storage fees for cold data. In addition to simplifying storage maintenance, Amazon S3 also supports versioning, which can help organize redundant backups based on timestamp.

- *Standard* - Amazon S3 Standard offers high durability, availability, and performance object storage for frequently accessed data. Because it delivers low latency and high throughput, Standard is perfect for a wide variety of use cases.
- *Infrequent Access* - Infrequent Access (Standard - IA) is an Amazon S3 storage class for data that is accessed less frequently but requires rapid access when needed. It offers high durability, throughput, and low latency like Amazon S3 Standard, with a low per GB storage price and per GB retrieval fee. This combination of low cost and high performance

makes it a sensible option for backups and as a data store for disaster recovery.

- *Archive* - AWS Glacier is a low-cost, long-term storage service that provides secure, durable storage intended for data backup and archival. AWS Glacier provides reliable, long-term storage for your data and eliminates the administrative burdens of operating and scaling storage to AWS. Using AWS Glacier, Neo4j backup operators never have to worry about capacity planning, hardware provisioning, data replication, hardware failure detection and repair, or time-consuming hardware migrations. Long-term storage on AWS Glacier is the least expensive storage tier per GB. However, the SLA for retrieving data has a much longer latency and is typically in the 3- to 5-hour range, whereas the other storage tiers have a shorter retrieval time measured in milliseconds.

By default, a consistency check is run at the end of each full backup to ensure that the files being moved to long-term storage will be usable upon recovery. The consistency checker is a relatively intensive operation since it makes a thorough check of the graph structure at the individual record level. So running the consistency checker on the same EC2 instances as your production cluster will result in performance degradation. For this reason, it is advisable to run this process on another instance. It favors EC2 instances with high I/O capacity and large RAM such as the i2.8xlarge instance. However, this instance doesn't need to be continuously active—it needs only to be instantiated for the duration of the backup and consistency check. Any failure during backup (such as the unscheduled termination of the underlying EC2 instance) means that the backup must be repeated.

After you have a full backup you can then take incremental backups against that state. An incremental backup is performed whenever an existing backup directory is specified, which the backup tool will automatically detect. The backup tool will then copy any new transactions from the Neo4j instance and apply them to the backup. The result will be an updated backup that is consistent with the current server state, and specifically is one that:

- Requires a full backup be completed first
- Replays logs of transactions since last backup

Restoring from a backup is very easy. This is an important operational affordance since restoring is typically done when a catastrophe has occurred. To restore, you simply do the following tasks:

1. Make sure Neo4j is not running.
2. Replace the `<Neo4j home>/graph.db` directory with the contents of the backup.
3. Start Neo4j. (If clustered, start the first instance and then rolling start the remaining instances.)

Now that you have seen the nuts and bolts of a Neo4j backup on AWS, you can focus on having the appropriate backup hygiene by following these recommendations:

- Take regular, periodic full backups with an I/O and RAM-optimized EC2 instance. Repeat these backups if they fail. Move the backup to Amazon S3 (or to Amazon EBS).
- Take incremental backups several times a day, ensuring the Neo4j log files are kept for longer than this period. Ensure that the backups are transmitted to Amazon S3 (or to Amazon EBS).

Disaster Recovery

After you have a Neo4j backup on stable long-term storage, disaster recovery (DR) is greatly simplified. If an incident occurs that, for whatever reason, wipes out all your active Neo4j instances and irrevocably wipes all instance storage, then you must quickly work to restore service.

Fortunately, DR with Neo4j on AWS is straightforward. You can place backups in long-term stable storage and restore them by a simple file-copy in the event of a disaster. From there you can seed a new cluster of Neo4j instances and resume service. Any transactions that occurred between the backup and disaster will have been lost.

Neo4j clusters can easily span multiple Availability Zones within the same VPC to create private, logically isolated networks. We recommend that you use a design for deploying Neo4j on multiple Availability Zones. ELB load balancers can operate across multiple Availability Zones, which enables these high availability designs to function seamlessly.

In addition to using a multiple Availability Zone design, it is also possible to use multiple Regions. One useful DR pattern is to host an instance or instances of Neo4j in other AWS Regions in slave and read-only mode. All slaves in Neo4j, whether they are read-write, read-only, or slave-only, are replicated asynchronously from the master. This asynchronous replication allows for regional diversity and availability of the database. Asynchronous replication across Regions is quite normal with Neo4j. However, typically one Region is designated as the master Region and other Regions are designated as slave Regions that only contain slave-only + read-only instances. In the extremely rare event of a regional failure, there is an administrative procedure to change one of the slave-only Regions to be the master.

It is important to note that slave and read-only instances never volunteer to take on important roles in the Neo4j HA cluster, but they are fed a stream of transactions from that cluster. This means that such instances can be used as a means of keeping a live backup of a cluster with a minimal downtime window between disaster and recovery. On disaster, we simply take the data store directory from one of the remote DR instances and seed a new cluster.

Conclusion

The AWS Cloud provides a unique platform for running Neo4j clusters at scale. With capacities that can meet dynamic needs, costs based on usage, and easy integration with other AWS services such as Amazon CloudWatch, AWS CloudFormation, Amazon EBS, and Amazon S3, the AWS Cloud enables you to reliably run Neo4j at full scale without having to manage the hardware yourself. By using AWS services to complement the Neo4j graph database, AWS provides a convenient platform for developing scalable, high-performance applications atop Neo4j.

Customers who are interested in deploying Neo4j Enterprise on AWS now have access to a broad set of services beyond Amazon EC2, such as Elastic Load Balancing, Amazon EBS, Amazon CloudWatch, and Amazon S3. The combination of these services enable the creation of a reliable, secure, cost-effective, and performance-oriented graph database.

Contributors

The following individuals and organizations contributed to this document:

- Justin De Castri, Solutions Architect, AWS
- David Fauth, Field Engineer, Neo Technology
- Ian Robinson, Engineer, Neo Technology
- Jim Webber, Chief Scientist, Neo Technology

Further Reading

In addition to the depth of high-quality information available on AWS, there are several books on Neo4j that can help you get started with the database:

Graph Databases (O'Reilly): full e-book version available for free at http://graphdatabases.com

Learning Neo4j (Packt): http://Neoj4.com/books/learning-Neo4j/

The Neo4j manual (http://Neo4j .com/docs/stable/) has a wealth of information about the Neo4j Cypher query language, the programmatic APIs, and operational surface.

Notes

1 Availability Zones are distinct geographical locations that are engineered to be insulated from failures in other Availability Zones. They use separate power grids, ISPs, and cooling systems, and they are placed on different fault lines and flood plains when possible. All of this separation and isolation is designed to deliver a level of protection from the failure of a single instance to the failure of an entire Availability Zone.

2 This may change in future versions of Neo4j. Distributed transaction processing, which is at the heart of Neo4j clustering, is a fast-moving area in computer science, and the Neo4j team is very much involved with developing novel protocols for future releases.

3 http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/QuickStartEC2Instance.html

[4] http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/EC2NewInstanceCWL.html

[5] https://neo4j.com/developer/kb/amazon-cloudwatch-configuration-for-neo4j-logs/