Escuela Técnica Superior de Ingeniería de Telecomunicación

Universidad Politécnica de Cartagena.

Trabajo de Fin de Estudios.

Estudio comparativo de bases de datos para entornos Cloud y Big Data.

Autor: Juan Carlos Sánchez García.

Directora: María Dolores Cano Baños.

Murcia, a 14 de Julio de 2021

A mi familia.

Por toda la paciencia, apoyo y confianza depositada en mí todos estos años.

Os quiero.

Tabla de contenidos.

Índice de imágenes y tablas.

Capítulo 1. Introducción.

1.1. Importancia de las bases de datos en el Big data.

¿Qué es el Big Data? El Big Data es un término que ha adquirido una notoria popularidad en los últimos años y dependiendo de a quien le formules esta pregunta, obtendrás una respuesta u otra.

Si buscas una definición en Internet encontrarás quienes dicen que el Big Data es un término meramente descriptivo para definir un gran volumen de información.

Otros dirán que el término va más allá de definir un volumen y también se le atribuyen propiedades como la variedad, la velocidad, veracidad y una lista interminable de propiedades más.

También hay quienes dicen que el término Big Data es en realidad, un concepto que abarca una cadena de procesos en los que se extrae una información con cierto valor a partir del análisis realizado a un conjunto masivo de datos.

Para mí, desde el punto de vista de un aspirante a ser un ingeniero telemático, el Big Data es un largo y tedioso proceso que abarca muchos procesos a su vez en distintas especialidades. Y que, en una línea temporal, empezaría desde el envío de un dato hacia el conjunto masivo, hasta la recaudación del último céntimo obtenido de los beneficios de vender la información extraída.

Pero el objetivo de este proyecto no es darle una definición a el Big Data.

Durante mi etapa como estudiante he tenido la oportunidad de realizar prácticas en una empresa en la cual se manejaban volúmenes de datos enormes. Estos datos se recopilaban de sensores que hacían capturas con una frecuencia inferior al segundo. Posteriormente se actualizaba la base de datos diariamente con los datos recopilados el día anterior.

Todos estos datos se utilizaban para hacer análisis predictivos mediante algoritmos de inteligencia artificial de manera que permitiera a la empresa detectar posibles fallas futuras en determinadas máquinas y solucionarlas a tiempo para evitar el coste de reparación de estas. Y fue durante estas prácticas cuando entendí la importancia que tiene elegir un buen sistema de bases de datos para agilizar tanto el proceso de ingesta de datos como el de análisis.

1.2. Resumen de proyecto.

Con la experiencia adquirida durante estas prácticas, puedo decir que, lo que más importa a la hora de elegir una buena base de datos es la velocidad con la que se procesan las consultas de escritura y lectura.

Cómo esté estructurada la base de datos también determinará su eficacia, sobre todo a la hora de analizar la información almacenada. Pero con este proyecto, se pretende estudiar y analizar lo anteriormente indicado.

Para ello, he creado un sistema que haciendo uso de instancias EC2 del servicio de Amazon Web Services, consigo simular una prueba de estrés a una base de datos con hasta mil usuarios realizando consultas de lectura o escritura simultáneamente.

Generalmente las bases de datos no relacionales, también conocidas como NoSQL, se dividen en cuatro familias distintas; basadas en almacenar documentos, basadas en almacenar clave-valor, basada en gráficos y basada en almacenar familias de columnas.

En este proyecto he analizado cuatro bases de datos, cada una perteneciente a una familia distinta. Estas bases de datos son MongoDB, Cassandra, Neo4j y Hbase.

Volviendo a la prueba de estrés, esta consta de dos usuarios bien diferenciados. El usuario al que he denominado *master* es el usuario que alberga la base de datos a analizar. El otro usuario al que he denominado *worker* es el encargado de estresar la base de datos enviando peticiones.

Para realizar la prueba, he creado dos tipos de scripts los cuales son ejecutados uno por el master y otro por los *workers* para cumplir sus respectivas funciones.

Finalmente, añadir que he estudiado el comportamiento de las bases de datos en tres escenarios distintos. Un *worker* estresando la base de datos, cien *workers* estresando la base de datos y mil *workers* estresando la base de datos.

En la siguiente sección resumiré brevemente qué se puede encontrar en cada uno de los capítulos de la memoria.

1.3. Descripción de capítulos.

Capítulo 2. Estructura del sistema y funcionamiento.

A lo largo de este capítulo veremos la estructura del sistema y su funcionamiento desde un punto de vista generalizado. Después veremos las pequeñas variaciones que se han realizado al sistema para cada una de las distintas bases de datos.

Capítulo 3. Metodología.

A lo largo de este capítulo veremos, por un lado, las herramientas utilizadas para el desarrollo del proyecto y para qué sirven.

Por otro lado, veremos paso a paso como se han instalado y configurado todos los equipos.

Finalmente, veremos con detalle los scripts desarrollados.

Capítulo 4. Resultados de los Stress Test.

En este capítulo veremos los resultados obtenidos de realizar las pruebas. También expondré unas opiniones personales sobre las bases de datos y analizaremos los resultados obtenidos.

Capítulo 5. Conclusiones.

Finalmente, en este capítulo haré una recopilación de las conclusiones finales del proyecto.

Capítulo 2. Estructura del sistema.

2.1. Estructura generalizada.

Dado que el propósito del proyecto es hacer un análisis comparativo de distintas bases de datos, es necesario que las pruebas de estrés que se apliquen a cada base sean, en medida de los posible, equivalentes.

Cada una de las bases de datos estudiadas siguen un modelo no relacional distinto. Por tanto, hay ligeras variaciones en cuanto a la estructura de sistema para cada caso, pero de ninguna forma, estas variaciones influyen a la hora de calcular las métricas.

Con motivo de explicar el sistema de una manera generalizada y facilitar su compresión, en esta sección se detallará la estructura global desde el punto de vista del modelo relacional de bases de datos.

Más adelante se detallarán las ligeras variaciones de cada una de las bases de datos.

A continuación, se muestra una imagen que representa todos los elementos del sistema.

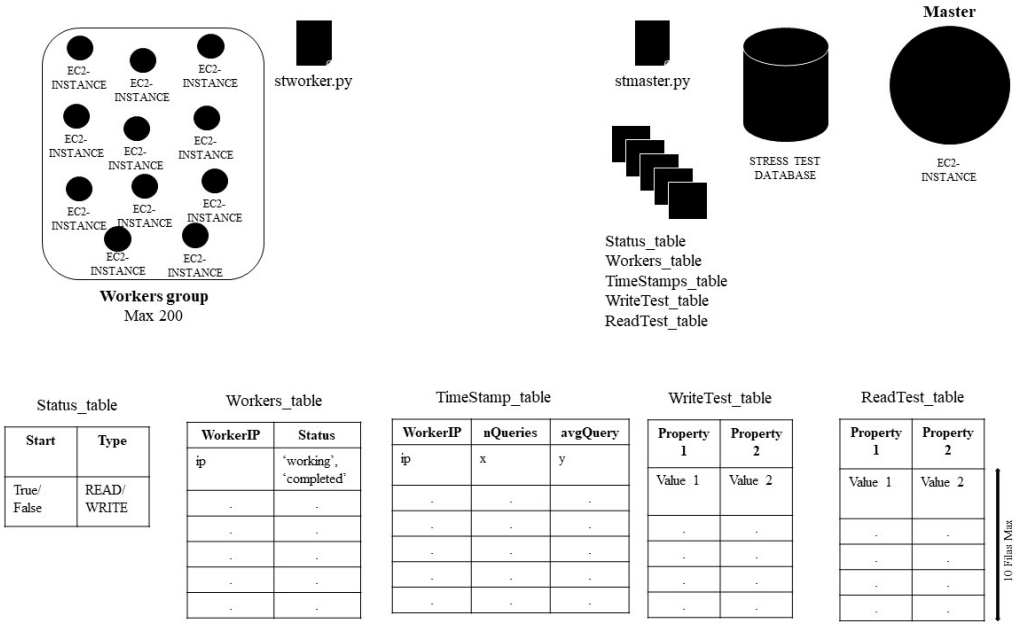

Imagen 1. Esquema general.

Como se puede apreciar en la imagen, se pueden distinguir dos tipos de usuarios representados por círculos, estos son:

- *Worker*: es el usuario encargado de ejecutar el script *stworker.py*.
- *Master*: es el usuario donde se alberga la base de datos instalada y desde donde se ejecuta el script *stmaster.py*.

Los scripts reciben una serie de argumentos al momento de ser ejecutados. En el caso del script *stworker.py* los argumentos son:

- -ip: con la dirección IP del usuario master.
- -s: tiempo en segundos que el usuario *worker* estará lanzando *queries* a la base de datos antes de finalizar la prueba.
- -t: número de *threads* que usará el *worker*. Este número corresponde al número de usuarios que se simularán dentro de una misma instancia. Esto se explicará con más detalle en capítulos posteriores.

Por otro lado, en la base de datos se encuentran cinco tablas previamente definidas.

- Status_table: está compuesta por dos columnas. Los valores permutarán durante la prueba con los valores que se pueden ver en la imagen. Siempre habrá datos en esta tabla.

- Workers_table: está compuesta por dos columnas y un número variable de filas dependiendo de la prueba. El objetivo de esta tabla es guardar la dirección IP de los usuarios *workers* y su estado. Inicialmente está vacía.

- TimeStamps_table: está compuesta por tres columnas y un número variable de filas dependiendo de la prueba. El objetivo es almacenar los valores de las métricas de cada usuario *worker* así como el número de métricas calculadas por cada uno durante la prueba. Cabe mencionar que este último dato se ha utilizado únicamente para el *debugging* durante el desarrollo de los scripts y no se toma en consideración a la hora de evaluar la base de datos. Inicialmente está vacía.

- WriteTest_table: esta tabla está compuesta por dos columnas e inicialmente ninguna fila. El objetivo de esta tabla es almacenar información durante la prueba de escritura. Inicialmente está vacía.

- ReadTest_table: esta tabla está compuesta por dos columnas y diez filas predefinidas con valores irrelevantes. El objetivo de esta tabla es almacenar la información que los usuarios leerán durante la prueba de lectura. Siempre habrá datos en esta tabla y nunca serán modificados.

Una vez definidos todos los elementos del sistema es el momento de ver el funcionamiento de este.

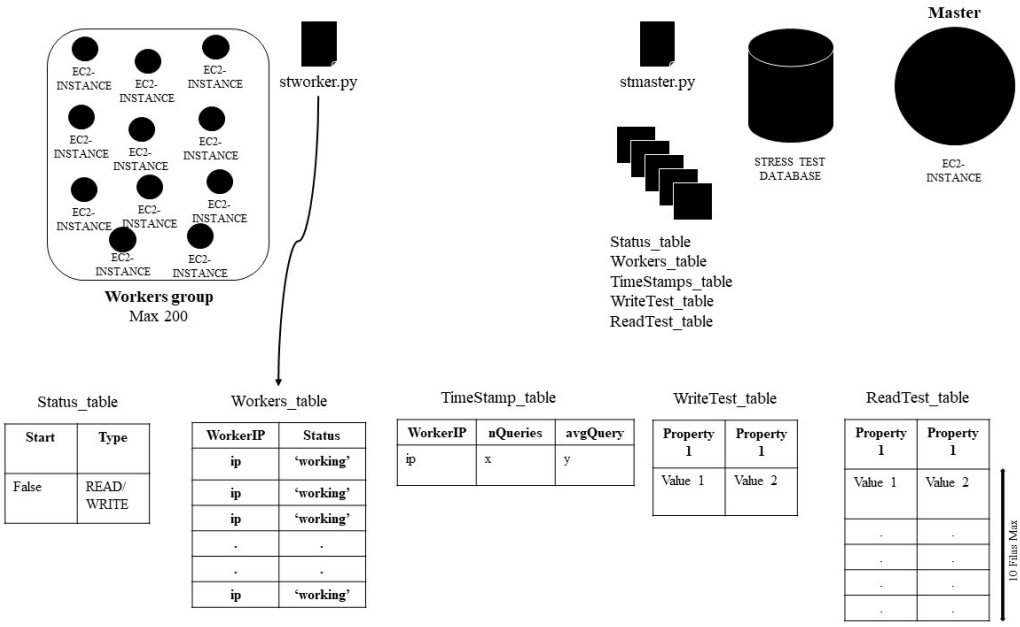

Imagen 2. Funcionamiento del sistema (I).

Primero los usuarios *workers* ejecutan el script *stworker.py* que actualizan las tablas Workers_table con su dirección IP y su estado actual, en este caso *working*. Una vez actualizada la tabla quedan a la espera de que el usuario master indique el comienzo de la prueba de estrés.

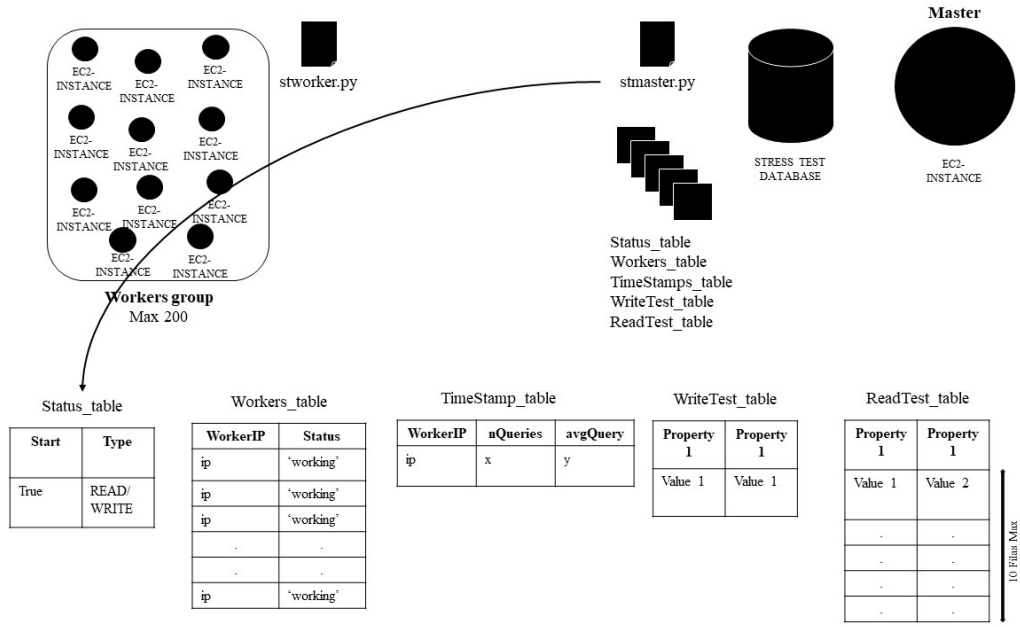

Imagen 3. Funcionamiento del sistema (II)

Después el usuario master ejecuta el script *stmaster.py* que actualiza la tabla Status_table, actualizando el valor de la columna Start a *True* y el valor de la columna Type a *Read* o *Write* en función del tipo de prueba que vaya a realizarse.

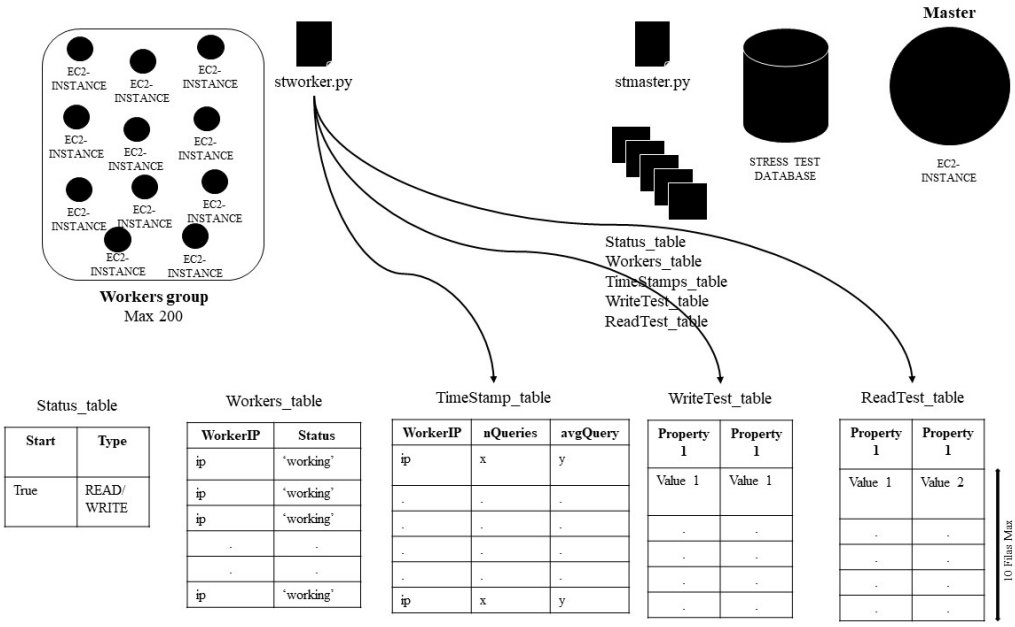

Imagen 4. Funcionamiento del sistema (III).

Cuando los usuarios *workers* leen el valor true de la tabla Status_table comienzan a inyectar filas en la tabla WriteTest_table (en el caso de la prueba de escritura) o a leer la tabla ReadTest_table (en el caso de la prueba de lectura). Todo esto durante el tiempo establecido al momento de ejecutar el script.

Durante este tiempo además de realizar las consultas a la base de datos, los *workers* calculan y guardan de forma local el tiempo de cada consulta.

Transcurrido este periodo, cada *worker* calcula una media de todas las métricas guardadas para posteriormente actualizar la tabla TimeStamp_table. Donde se registra la métrica de cada *worker*.

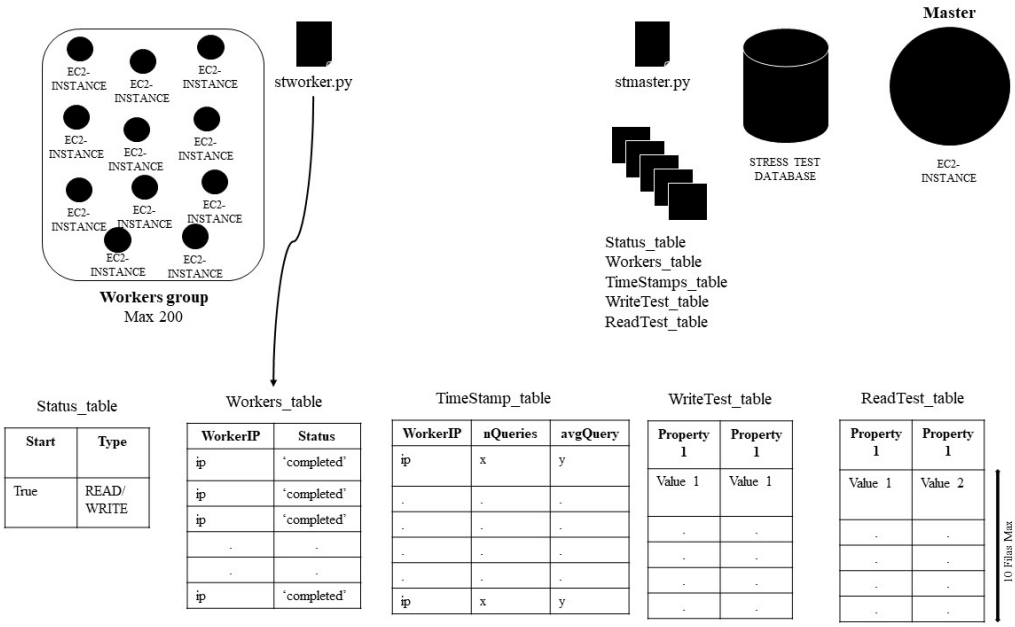

Imagen 5. Funcionamiento del sistema (IV).

Después cada usuario worker actualiza la tabla Workers_table actualizando el valor de la columna Status en la fila que insertó anteriormente con el valor *completed*.

Una vez hecho esto el worker finaliza la ejecución del script stworker.py.

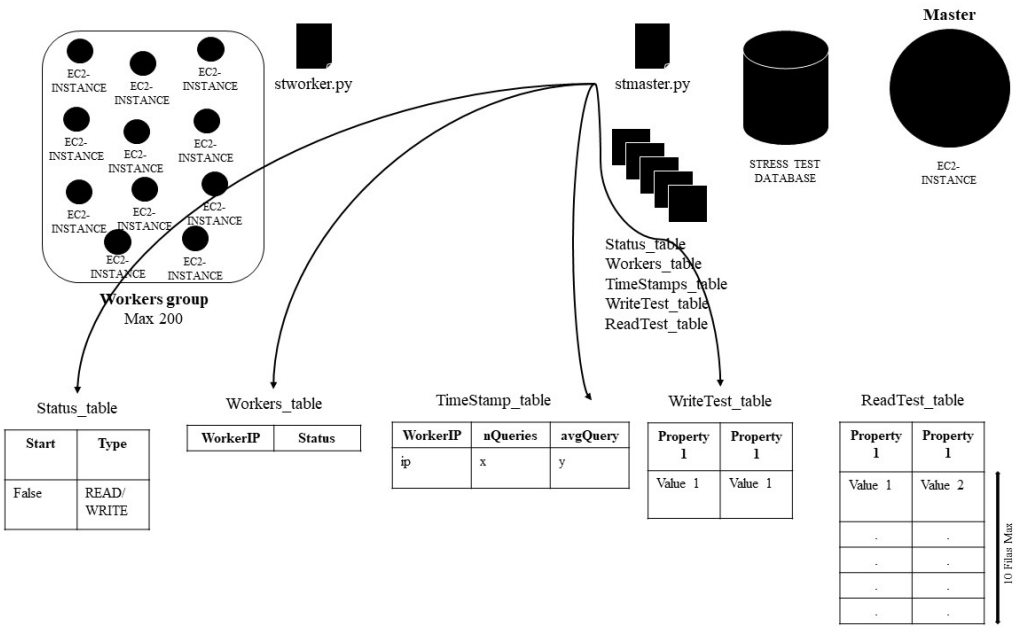

Imagen 6. Funcionamiento del sistema (V).

Finalmente el usuario master lee de la tabla TimeStamps_table la columna avgQuery y calcula la media de todas las filas. Obteniendo así el tiempo promedio de todos los usuarios worker. Despues limpia de datos la tabla.

También limpia de datos la tabla Workers_table y la tabla WriteTest_table (que se encontrará vacía o no dependiendo de si la prueba realizada ha sido de lectura o escritura).

Por último, actualiza el valor de la columna Start de la tabla Status_table a False nuevamente.

Antes de finalizar el script *stmaster.py* muestra por pantalla el resultado obtenido.

Esta es la estructura y el funcionamiento general que se ha replicado para todas las bases de datos. En el próximo apartado se detallarán las ligeras variaciones mencionadas anteriormente de cada una de las bases de datos.

2.2. Estructura MongoDB.

MongoDB[1] es la base de datos no relacional elegida dentro de la familia de bases de datos NoSQL basada en documentos.

Antes de comenzar con las variaciones respecto a la estructura generalizada del sistema es conveniente entender la equivalencia entre los elementos de MongoDB con los elementos de otras bases de datos relacionales.

En esta tabla se pueden ver las equivalencias:

Tabla 1. Equivalencia de elementos entre MongoDB y modelo relacional

Elemento modelo relacional	Elemento MongoDB
Base de datos	Base de datos
Tabla	Colección
Fila	Documento
Columna	Campo

MongoDB no se aleja mucho del modelo de bases de datos relacional. Su estructura es muy simular. De hecho, como se puede ver en la tabla, hay un elemento equivalente en MongoDB que cumpla una función similar a los elementos utilizados en el apartado anterior.

Es por ello que no han sido necesarias muchas modificaciones del sistema para realizar la prueba a esta base de datos.

La única variación en el sistema que ha sido necesaria afecta a la estructura de la tabla Status_table:

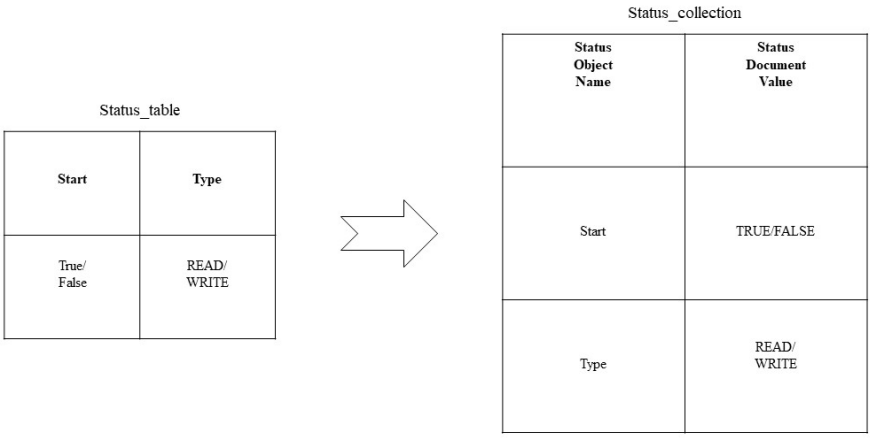

Imagen 7. Cambio en la estructura del sistema en MongoDB.

El motivo de este cambio es para facilitar la búsqueda de los valores *True/False* y *Read/Write* a la hora de desarrollar los scripts. Ya que uno de los parámetros a pasar en la función de búsqueda debe ser un valor del documento (fila). Esto se encuentra explicado con más detalle en la sección *3.3.1. MongoDB.*

2.3. Estructura Cassandra.

Cassandra[2] es la base de datos de la familia de bases de datos NoSQL basada en clave-valor. Aunque también tiene características de la familia basada en familias de columnas.

Esta base de datos también posee los mismos elementos que otras bases de datos relacionales. Incluso tiene un lenguaje de bastante pulido similar al SQL. Con un montón de agregaciones disponibles.

Para el propósito de este proyecto no se requieren cambios en la estructura. Aunque, con motivo de agilizar el desarrollo de los scripts, se ha cambiado la estructura de la tabla Status_table de la misma forma que en MongoDB.

2.4. Estructura de Neo4J.

Neo4J[3] pertenece a la familia de bases de datos no relacionales gráficas. Este tipo de modelo es el más alejado al modelo relacional.

Esta base de datos en particular no comparte ningún elemento del sistema generalizado, por tanto, intentar representar su estructura empleado los mismos elementos es una tarea complicada.

Neo4J no dispone de tablas, por tanto, tampoco de filas o columnas. La información se almacena en nodos. Estos nodos poseen propiedades y la forma de clasificar los nodos, de manera que a la hora de hacer una lectura no se lean todos los nodos de la base de datos, es mediante los llamados *labels* o etiquetas.

De modo que para extrapolar la estructura generalizada a este modelo de bases de datos se han requerido cinco tipos de nodos distintos, que son:

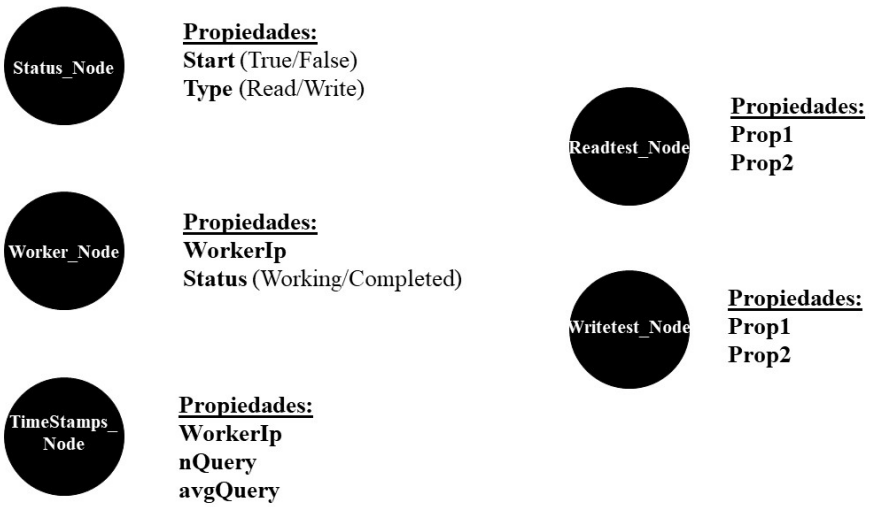

Imagen 8. Tipos de nodos empleados en Neo4J.

El funcionamiento del sistema es el mismo. Al comienzo de cada prueba se encontrarán creados once nodos, uno de tipo Status_Node y diez de tipo Rreadtest_Node. El resto se van creando y eliminando con el transcurso de la prueba.

De modo que cuando añadíamos filas a una tabla en el modelo generalizado, con Neo4J se crean nuevos nodos de un tipo específico.

2.5. Estructura de Hbase.

Hbase[4] es la base de datos correspondiente a la familia de bases de datos NoSQL basadas en familia de columnas.

Este modelo de base de datos, como las dos anteriores MongoDB y Cassandra, poseen elementos muy parecidos a las bases de datos relacionales.

No se ha requerido ninguna modificación en la estructura del sistema generalizado.

Capítulo 3. Metodología.

3.1. Herramientas de desarrollo.

Durante esta sección se presentarán todas las herramientas utilizadas para la elaboración del proyecto.

A modo de resumen, este es el listado completo:

- Instancias EC2 de AWS.
    - Instancias t2.micro.
    - Instancias t2.2xlarge.
- Servicio Run Command de AWS.
- Mongo Compass.
- HBase
- Anaconda Navigator.
    - Spyder.
    - CMD.exe Prompt.
- Python 3.7.
    - Librería Pymongo.
    - Librería Cassandra.
    - Librería Neo4J.
    - Librería Argparse.
    - Librería Request.
    - Librería Statistics.
    - Librería Datetime.
    - Librería Concurrent.Futures
    - Librería Random.
    - Librería Numpy.
- Protocolo SSH.
- Protocolo SCP.

La prueba de estrés requiere un número elevado de equipos. Dado que físicamente ha sido imposible encontrar mil equipos se ha utilizado el servicio de Amazon Web Services (AWS) que permite alquilar equipos en la nube. Este servicio se llama Elastic Computing Cloud conocido como EC2.

Los usuarios del sistema (*workers* y *master*) utilizan instancias de este servicio. El tipo de instancia son “t2.micro” para las workers y “t2.2xlarge” para las masters.

Las instancias tipo t2.micro tienen 1 vCPU y 2 GiB de memoria. Las instancias tipo t2.2xlarge tienen 8 vCPU y 36 GiB de memoria. Las características hardware completas se pueden encontrar en el catálogo oficial de AWS [5].

Un punto a aclarar sobre los tipos de instancias elegidos es que, no todas las bases de datos requieren las mismas prestaciones. Pero para garantizar un entorno de trabajo justo a la hora de hacer las pruebas se ha utilizado el mismo tipo para todas.

Para ejecutar los scripts desde todas las instancias se ha utilizado el servicio Run Command. Este servicio permite la ejecución simultánea de los scripts.

Durante el desarrollo de los scripts para MongoDB, he utilizado el navegador Mongo Compass que facilita la visualización de la base de datos.

Para el desarrollo de scripts se ha utilizado el entorno de desarrollo Anaconda Navigator el cual permite crear entornos de trabajo y enlazar aplicaciones a estos entornos. En concreto se ha creado un entorno con Python 3.7. La aplicación usada para escribir los scripts es Spyder. También se ha usado la aplicación de terminal CMD.exe Prompt que permite ejecutar comandos de consola desde el entorno creado anteriormente.

Para la comunicación directa con los terminales de las instancias se ha usado el protocolo SSH.

Para el intercambio de ficheros se ha utilizado el protocolo SCP.

3.2. Configuración de los equipos.

El servicio AWS permite la creación de imágenes virtuales de las instancias para posteriormente crear nuevas instancias a partir de estas imágenes. Estas imágenes se llaman técnicamente Amazon Machine Image (AMI).

Para el desarrollo del proyecto primero he creado una instancia base instalando Python3.7 y guardando la AMI.

A partir de esta AMI he ido creando y configurando el resto de instancias. Conforme terminaba de instalar y configurar he ido guardando las AMIs de cada una de ellas.

A lo largo de este apartado detallaré todas las configuraciones que hay que hacer previamente en AWS para crear las instancias, cómo he creado y configurado las instancias y también cómo usar las herramientas utilizadas tanto para el envío de ficheros como para el acceso remoto a las instancias.

3.2.1. Configuración de AWS.

Antes de crear instancias para los usuarios es necesario crear un par de claves para la comunicación remota a través del protocolo SSH.

Para ello hay que ir al panel de AWS EC-2, en el apartado Network & Security > Key Pairs. Como el sistema está compuesto por dos usuarios bien diferenciados he creado dos pares de claves, uno para cada tipo.

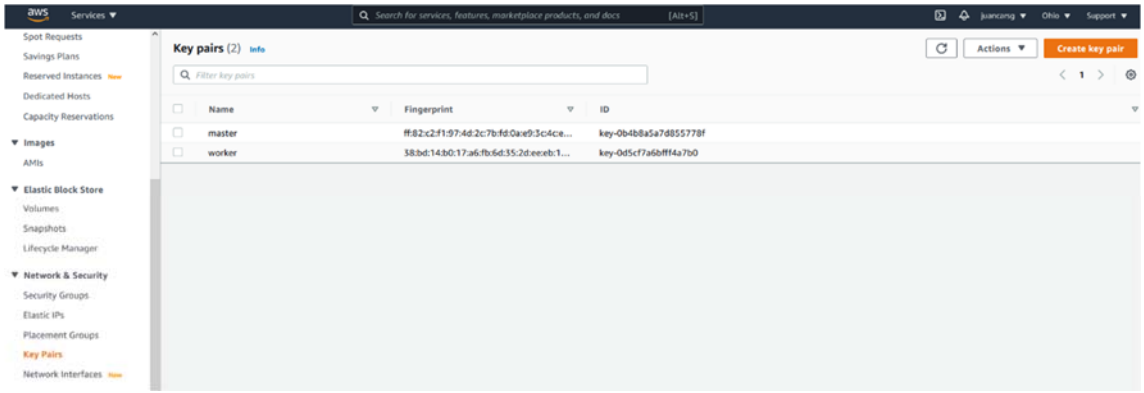

Imagen 9. Pares de claves usados en el sistema.

Una vez creados los pares de claves hay que crear un rol para la dirección de las instancias. Para ello hay que seguir todos los pasos de la guía oficial de AWS [6].

Siguiendo la guía, primero hay que acceder a la consola de AWS System Manager y seguir los siguientes pasos:

- Ir a apartado Quick Setup.

- Seleccionar botón Create.
- Elegir opción Host Management.
- Dejar todas las opciones por defecto en el apartado de configuración. Importante asegurarse de que en el apartado Targets, esté seleccionada la opción All Instances para que en un futuro al crear una instancia podamos seleccionar este rol.

Después de esto ya estaría creado este rol.

También es necesaria la creación de un segundo rol para permitir a las instancias utilizar el servicio Run Command.

Para la creación de este rol hay que seguir los pasos de la guía oficial de AWS[7].

Siguiendo la guía hay que acceder a la consola de IAM. Una vez dentro de esta aplicación hay que seguir los siguientes pasos:

- En el panel de navegación, elegir Roles y seleccionar Create Role.
- Donde dice Select type of trusted entity elegir AWS service.
- Debajo elegir EC2.
- En los permisos elegir AmazonEC2RoleforSSM.
- Nombrar como EnablesEC2ToAccessSystemsManagerRole.

Finalmente, ya estarían creados los dos roles a usar.

3.2.2. Configuración de instancia base.

Para crear una instancia hay que seleccionar la opción Launch Instances. Después se entra en el proceso de configuración de la instancia y hay que seguir los siguientes pasos:

- Seleccionar una Amazon Machine Image (AMI). Todas las instancias configuradas en este proyecto se han basado en la Amazon Linux 2 AMI.

- Seleccionar el tipo de instancia. Como se trata de la instancia base a partir de la cual luego crearemos otras, se ha utilizado un tipo con bajas prestaciones ya que únicamente se usará para instalar herramientas y configurar el equipo. En este caso de tipo t2.micro.

- Seleccionar un IAM Role. Dependiendo del usuario destinado a la instancia usaremos AmazonSSMRoleForInstanceQuickSetUp para el caso del usuario master o EnablesEC2ToAccessSystemsManagerRole para el caso del usuario *worker*. Para la instancia base se necesita el primero.

- El resto de configuraciones se han dejado por defecto.

- Finalmente te pedirá qué par de claves usar. Elegir una de las dos opciones creadas en el anterior apartado dependiendo del tipo de usuario.

Una vez creada la instancia base hay que instalar las herramientas necesarias.

Existen dos formas de acceder remotamente a la instancia:

1. A través de la aplicación de AWS EC2, simplemente haciendo *click* derecho encima de la instancia (encendida) y dándole a *connect*.

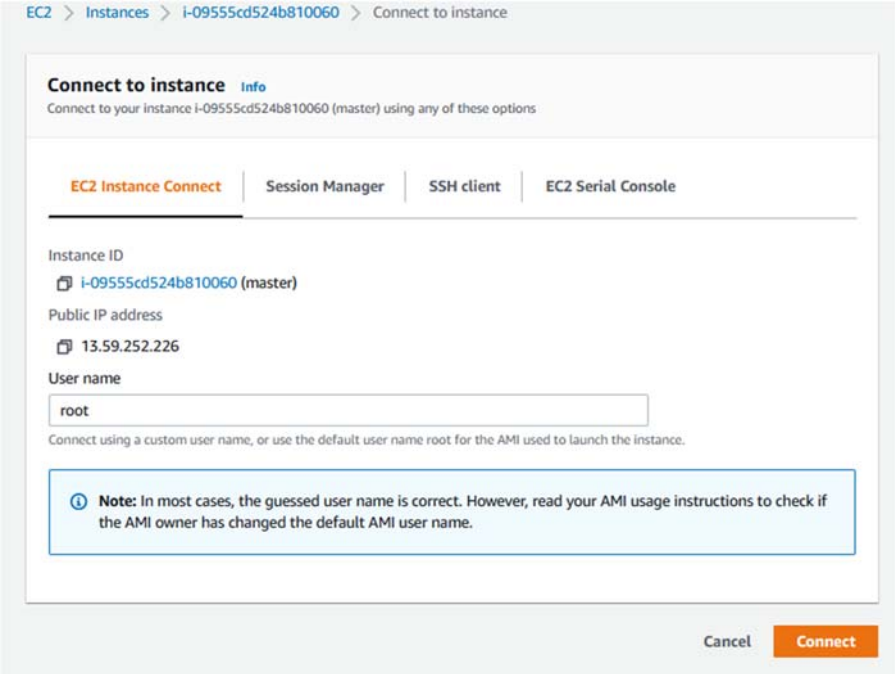

Imagen 10. Acceso remoto a instancia a través de la aplicación AWS EC2.

2. A través de consola utilizando el protocolo SSH.

   Esta opción es la que más he utilizado durante el desarrollo del proyecto ya que con la anterior, al tratarse de una aplicación web, en algunas ocasiones su rendimiento no era el óptimo.

   Para conectarse hay que realizar los siguientes pasos:

   1. Abrir una consola.
   2. Ejecutar el siguiente comando:

```
ssh -i path_to_the_file/instance.pem ec2-user@ip-address
```

   3. Escribir *Yes* al aparecer el mensaje.

   En mi caso un ejemplo sería:

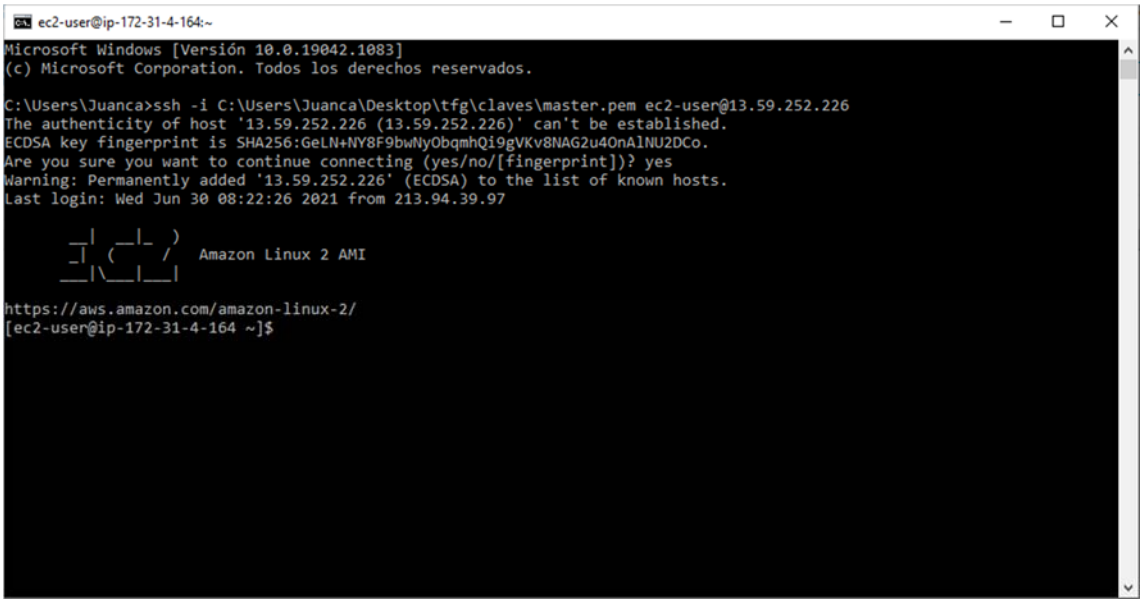

Imagen 11. Ejemplo acceso remoto mediante el protocolo SSH a una instancia.

Para instalar Python3.7 una vez dentro de la instancia hay que seguir los siguientes pasos:

1. Entrar en modo *superuser* con:

```
sudo su -s
```

2. Instalar el compilador GCC:

```
yum install gcc openssl-devel bzip2-devel libffi-devel
```

3. Descargar el paquete:

```
sudo wget https://www.python.org/ftp/python/3.7.9/Python-3.7.9.tgz
```

4. Descomprimir el paquete:

```
sudo tar xzf Python-3.7.9.tgz
```

5. Entrar en el directorio e instalar:

```
cd Python-3.7.9
sudo ./configure --enable-optimizations
sudo make altinstall
```

6. Borrar el paquete

```
cd -
sudo rm /usr/src/Python-3.7.9.tgz
```

7. Verificar instalación:

```
Python3.7 -V
```

Una vez instalado Python hay que instalar la herramienta Pip. Esta herramienta es muy útil para instalar en un futuro los módulos que necesiten los scripts. Para ello hay que seguir los siguientes pasos:

1. Descargar el script de instalación:

```
curl -O https://bootstrap.pypa.io/get-pip.py
```

2. Ejecutar el script.

```
python3.7 get-pip.py --user
```

3. Añadir la ruta del ejecutable a la variable $PATH.

```
export PATH=LOCAL_PATH:$PATH
source ~/.bashrc
```

4. Verificar instalación.

```
pip –version
```

Una vez instalado Python3.7 y la herramienta Pip, es el momento de guardar la AMI de la instancia.

Para ello lo más recomendable es apagar la instancia actual. Para ello basta con hacer *click* derecho encima de la instancia en el panel de AWS EC2 y darle a la opción *Stop instance*. Después de la misma forma, darle a la opción *Create image* en *Images and templates*.

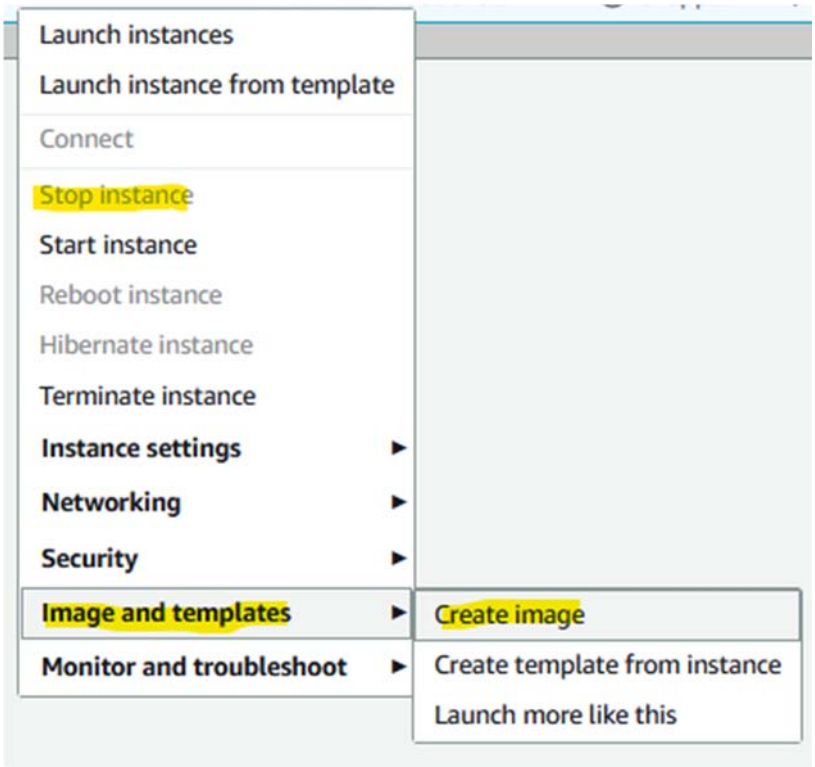

Imagen 12. Creación de AMI de una instancia.

Después se abrirá una ventana de configuración donde hay que dejarlo todo por defecto y darle un nombre a la imagen.

A partir de aquí ya existiría una AMI a partir de la cual crear el resto de instancias.

3.2.3. Configuración de instancias worker.

Los workers no necesitan ninguna otra herramienta más que Python3.7. Lo único a tener en cuenta es que, para cada script se requieren distintos módulos. Por tanto, de cara a realizar las pruebas, habrá que arrancar una instancia worker, enviarle mediante el protocolo SCP el script que se tenga que ejecutar e instalar los módulos necesarios para la ejecución de dicho script.

Para enviar el script hay que abrir una consola y ejecutar el siguiente comando:

```
scp -i local_path_to/worker.pem local_path_to/stworkerSCRIPT.py ec2-
user@<ip address instance>:/home/ec2-user/scripts/stworkerSCRIPT.py
```

Este comando nos copiará el script en el directorio scripts del usuario ec2-user.

Para instalar los módulos que requiere el script únicamente hay que usar el script Pip, previamente instalado, haciendo uso del comando:

```
pip install <module_name>
```

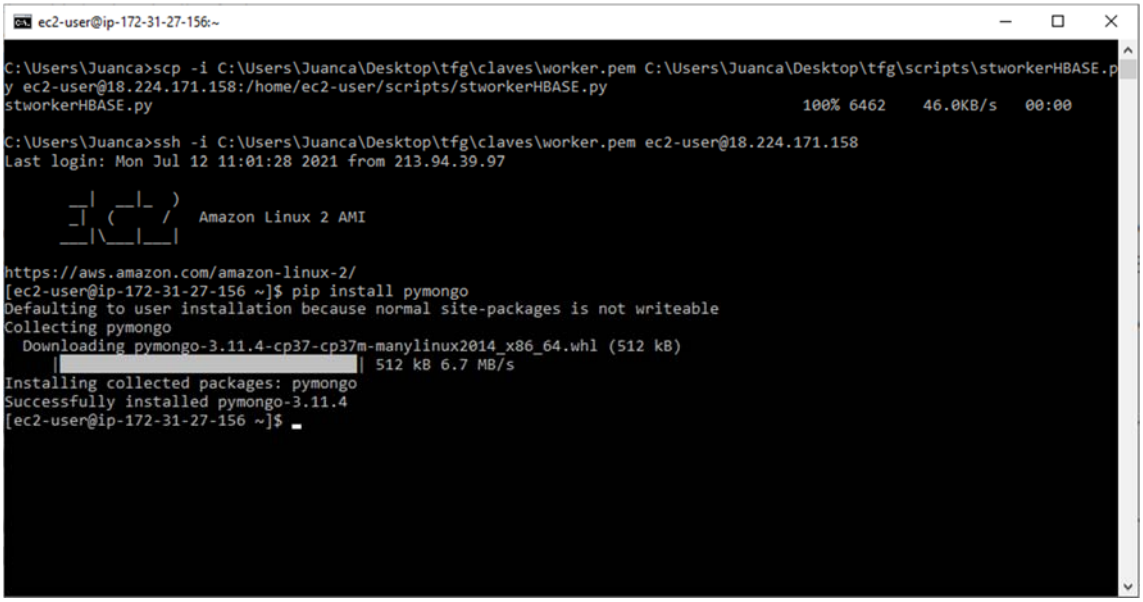

Imagen 13. Ejemplo uso protocolo SCP y comando PIP.

3.2.4. Configuración de MongoDB.

Ya hemos visto en anteriores secciones cómo crear una instancia. Para crear las instancias de tipo master hay que cambiar dos opciones, que son:

- A la hora de seleccionar AMI, hay que elegir la AMI de la instancia base creada con anterioridad.
- A la hora de seleccionar el tipo de instancia, hay que elegir t2.2xlarge.

Antes de comenzar con la instalación de MongoDB hay que añadir reglas a la instancia para permitir que entre tráfico TCP por el puerto 27017, que es el que usa MongoDB.

Para ello hay que:

1. Hacer Click en detalles de la instancia.
2. Ir al apartado de Security.
3. Hacer click en el enlace debajo de Security groups.
4. Pulsa en Edit Inbound Rules.
5. Añadir las siguientes reglas:

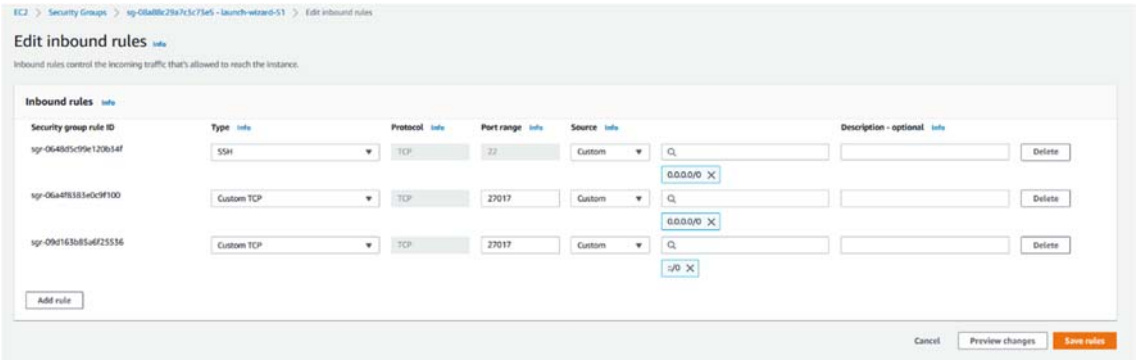

Imagen 14. Agregar Inbound Rules para habilitar los puertos de una instancia.

La primera regla viene por defecto en todas las instancias.

A partir de este momento la instancia tendrá habilitada la entrada de tráfico a través del puerto que usa MongoDB y ya se puede comenzar a instalar la base de datos.

Una vez dentro de la instancia ya sea a través de la aplicación de AWS o a través de SSH, los pasos para instalar MongoDB siguiendo la guía oficial de MongoDB [8] son los siguientes:

1. Crear el repositorio de Yum:

```
Cd /etc/yum.repos.d/
Sudo nano mongo.repo
[mongodb-org-4.4]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/amazon/2/mongodb-
org/4.4/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-4.4.asc

(Usar ctrl + X y escribir Y para salir de nano y guardar)
```

2. Instalar MongoDB con yum:

```
sudo yum install -y mongodb-org
```

3. Si se quiere habilitar el arranque de inicio (opcional):

```
sudo systemctl enable mongod
```

4. Arrancar MongoDB:

```
sudo systemctl start mongod
```

En este punto ya estaría MongoDB completamente instalada. Para la ejecución del proyecto no se ha requerido modificar ningún archivo de configuración.

Lo siguiente es crear y configurar las bases de datos con las colecciones necesarias para la prueba.

Esto puede hacerse de varias formas:

- A través de la consola de MongoDB dentro de la instancia.
- Mediante un script de configuración usando la librería *pymongo*.
- A través de la aplicación Mongo Compass.

Mongo Compass es una aplicación que te permite interactuar con las bases de datos de una manera muy sencilla y visual. Es por ello, que en el caso de Mongo DB, la creación y configuración de bases de datos la he realizado mediante esta herramienta, que puede conectarse a la base de datos de manera remota.

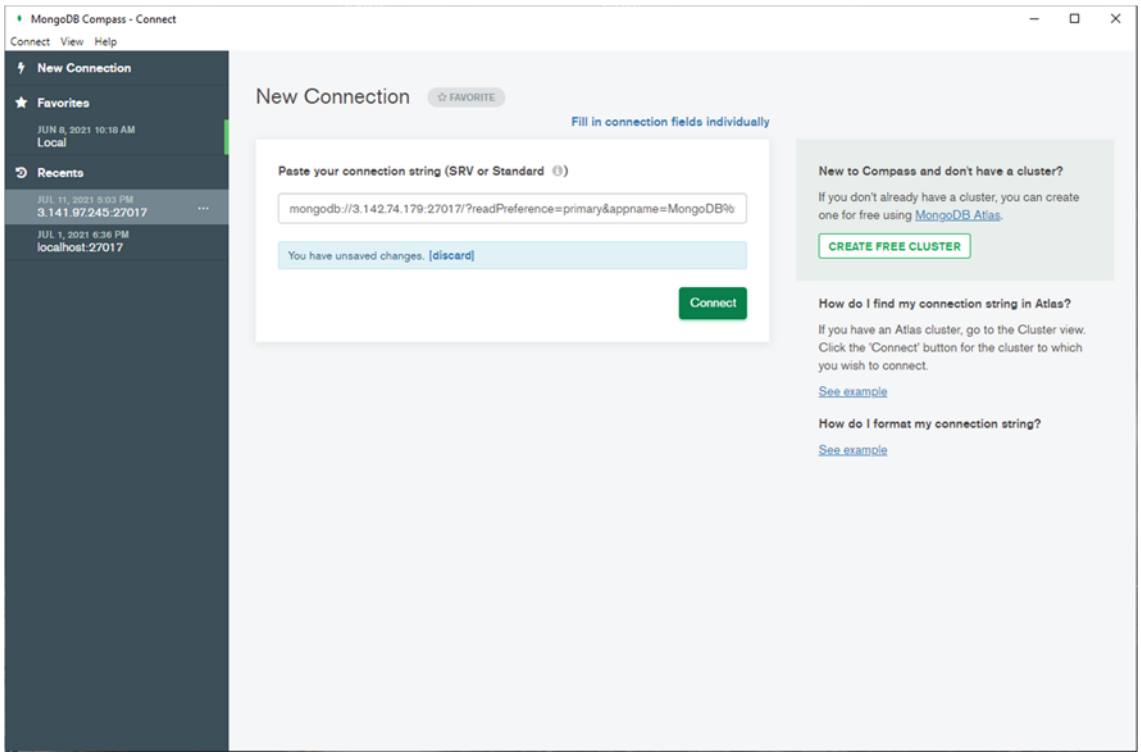

Imagen 15. Cómo conectarse a la base de datos a través de Mongo Compass.

Como se puede apreciar en la figura, para acceder a las bases de datos hay que crear una nueva conexión y editar el campo de texto y poner la dirección IP de la instancia en lugar de localhost.

Una vez dentro se visualizaran las bases de datos existentes. En un principio se verán tres que vienen por defecto; admin, config y local.

Para crear bases de datos hay que pulsar el boton Create database. Acto seguido pedirá un nombre para la base de datos y un nombre para una colección.

Para desarrollar las pruebas he creado dos bases de datos. Una llamada status y otra llamada test.

En status he creado tres colecciones; status_collection, timeStamps y workers.

En test he creado dos colecciones; testread y writetest.

Estas colecciones corresponden a las tablas necesarias para el funcionamiento del sitema mencionadas en la sección *2.1. Estructura generalizada.*

Para el funcionamiento del sistema hay que añadir los documentos necesarios en las colecciones status_collection y readtest.

Para ello se ha realizado vía script de Python usando la librería PyMongo.

Estos scripts realizan una serie de consultas de escritura para insertar los documentos necesarios. Este es el resultado:

status_collection

	_id ObjectId	status String	statusObjectName String
1	60bf31bf2123c370d9721f35	"false"	"start"
2	60c0f243d691cafd41f2d022	"write"	"type"

Imagen 16. Documentos insertados en la colección status_collection.

readtest

	_id ObjectId	id String	resultado String
1	60e5c76a7f89ba646ebcf86b	"0"	"2-6"
2	60e5c76a7f89ba646ebcf86c	"1"	"2-6"
3	60e5c76a7f89ba646ebcf86d	"2"	"2-6"
4	60e5c76a7f89ba646ebcf86e	"3"	"2-6"
5	60e5c76a7f89ba646ebcf86f	"4"	"2-6"
6	60e5c76a7f89ba646ebcf870	"5"	"2-6"
7	60e5c76a7f89ba646ebcf871	"6"	"2-6"
8	60e5c76a7f89ba646ebcf872	"7"	"2-6"
9	60e5c76a7f89ba646ebcf873	"8"	"2-6"
10	60e5c76a7f89ba646ebcf874	"9"	"2-6"

Imagen 17. Documentos insertados en la colección readtest.

En MongoDB no es necesario predefinir los campos que van a usar los documentos por tanto no es necesaria ninguna otra acción en el resto de colecciones.

Con esto MongoDB, ya está completamente configurado para las pruebas de estrés.

3.2.5. Configuración de Cassandra.

Al igual que con MongoDB, al crear la instancia hay que seleccionar la AMI de la instancia base y el tipo de instancia t2.2xlarge.

Unos de los requisitos de Cassandra es tener instalado Java JDK 8.

Para instalar JDK 8 hay que seguir los siguientes pasos:

1. Instalar Java JDK 8 con Yum. (No es necesario crear un repositorio ya que viene de base).

   ```
   sudo yum install java-1.8.0-openjdk.x86_64
   ```

2. Comprobar que se ha instalado correctamente:

   ```
   java -version
   ```

3. Ver localización de instalación:

   ```
   file /etc/alternatives/java
   ```

4. Establecer la variable JAVA_HOME:

   ```
   Sudo nano /home/.bashrc

   export JAVA_HOME="/usr/lib/jvm/jdk-1.8.0-openjdk.x86_64"

   PATH=$JAVA_HOME/bin:$PATH
   (ctrl+x para salir de nano)

   source .bashrc

   export PATH=$PATH:$JAVA_HOME/bin
   ```

Este requisito también lo necesita Hbase por lo que he actualizado la AMI base antes de seguir con la instalación de Cassandra y en un futuro no tener que volver a repetir este proceso.

Una vez actualizada la AMI base, antes de seguir con la instalación también hay que añadir las Inboud Rules para habilitar el puerto que usa Cassandra, en este caso 9042.

Para instalar Cassandra hay que seguir los siguientes pasos:

1. Crear el repositorio de Yum:

   ```
   sudo nano /etc/yum.repos.d/datastax.repo
   [datastax]
   name = DataStax Repo for Apache Cassandra
   baseurl = http://rpm.datastax.com/community
   ```

```
enabled = 1
gpgcheck = 0
```

2. Instalar el paquete:

```
sudo yum install dsc20-2.0.11-1 cassandra20-2.0.11-1
```

3. Si se quiere habilitar el arranque de inicio (opcional):

```
sudo service cassandra enable
```

4. Arrancar Cassandra:

```
sudo service cassandra start
```

En este punto ya está instalada Cassandra.

A diferencia de MongoDB, Cassandra no dispone de una aplicación similar a Mongo Compass, por lo que para configurar la base de datos hay hacerlo a través de la consola del servidor.

Para entrar en esta consola hay que escribir el siguiente comando:

```
Cqlsh -u cassandra -p cassandra
```

Cassandra en vez de crear bases de datos crea Keyspaces.

Para crear un Keyspace hay que escribir el siguiente comando:

```
CREATE KEYSPACE stresstest WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 1 };
```

El parámetro class se refiere a la forma en la que Cassandra almacena los datos.

El modelo de datos que propone Cassandra está pensado para implementar varios nodos que conformen un *cluster* por lo que permite que los datos se repliquen entre nodos.

Como en este proyecto esto no va a ser necesario se ha seleccionado la SimpleStrategy y el factor de réplica a uno.

Una vez creado el Keyspace hay que crear las tablas que, a diferencia de MongoDB, sí que hay que definir las columnas y tipo de valores de estas.

Para crear las cinco tablas necesarias para el funcionamiento del sistema hay que ejecutar los siguientes comandos:

```
Use stresstest;
```

```
create table status_table (statusObjectName VARCHAR PRIMARY KEY,
status VARCHAR);

create table workers (workerIP VARCHAR PRIMARY KEY, threadID VARCHAR,
status VARCHAR);

create table timeStamps (ip VARCHAR PRIMARY KEY, nQuery VARCHAR,
avgQuery VARCHAR);

create table readtest (id VARCHAR PRIMARY KEY, value VARCHAR);

create table writetest (id VARCHAR PRIMARY KEY, value VARCHAR,
avgQuery VARCHAR);
```

Una vez creadas todas las tablas se pueden ver usando el commando:

```
Describe tables;
```

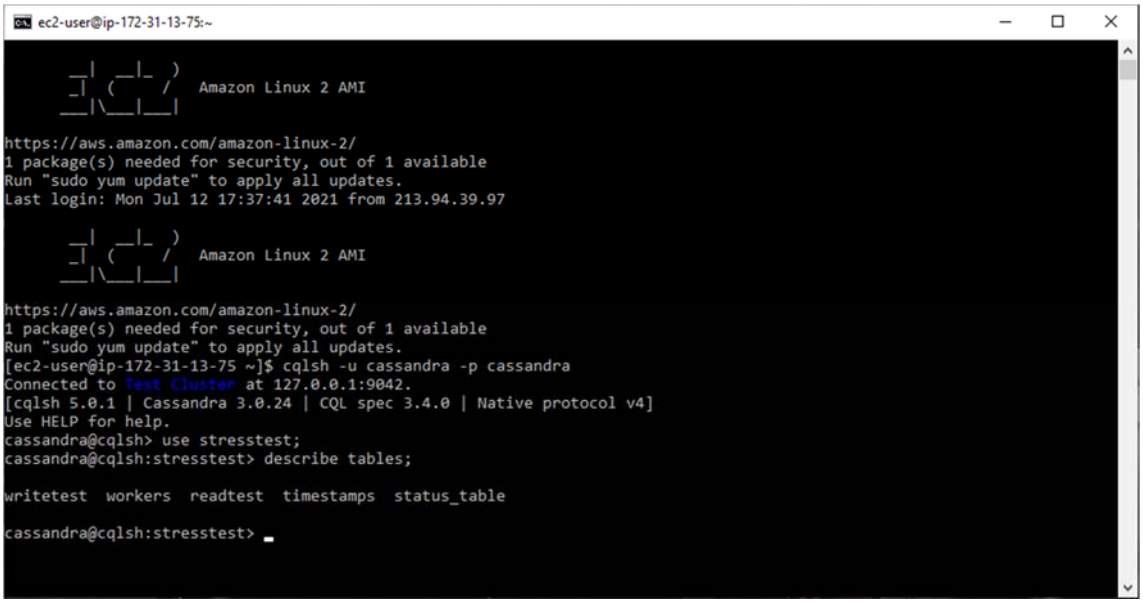

Imagen 18. Tablas creadas en Cassandra.

Una vez creadas las tablas hay que añadir las filas necesarias en la tabla status_table y readtest. Para ello hay que ejecutar los siguientes comandos:

```
insert into stresstest.status_table (statusObjectName, status) values
('start','false');

insert into stresstest.status_table (statusObjectName, status) values
('type','read');

insert into stresstest.readtest (id, value) values ('1','value');

insert into stresstest.readtest (id, value) values ('2','value');

insert into stresstest.readtest (id, value) values ('3','value');

insert into stresstest.readtest (id, value) values ('4','value');

insert into stresstest.readtest (id, value) values ('5','value');

insert into stresstest.readtest (id, value) values ('6','value');

insert into stresstest.readtest (id, value) values ('7','value');
```

```
insert into stresstest.readtest (id, value) values ('8','value');
insert into stresstest.readtest (id, value) values ('9','value');
insert into stresstest.readtest (id, value) values ('10','value');
```

Cabe mencionar que en casandra siempre debe de haber una columna que sea clave primaria. Por tanto, en estos casos no puede haber duplicidad de valores en ninguna de las filas.

Para ver los detalles de las tablas se puede hacer uso del comando:

```
Describe table <table name>
```

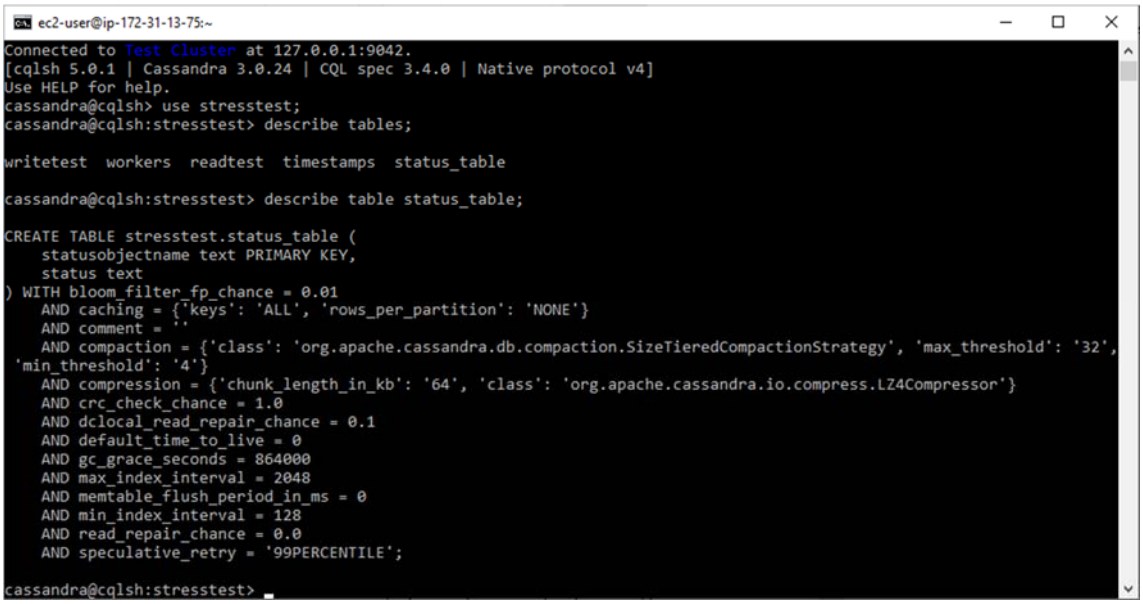

Imagen 19. Ejemplo de uso del comando describe table.

Con esto Cassandra, ya está completamente configurada para las pruebas de estrés.

3.2.6. Configuración de NeoJ4.

Como ya se ha mencionado en secciones anteriores, a la hora de crear la instancia hay que seleccionar la AMI base y el tipo de instancia t2.2xlarge.

Además, hay que habilitar dos puertos de la misma forma que con las anteriores bases de datos. Los puertos son 7474 y 7687.

Para instalar NeoJ4 hay que seguir los siguientes pasos según la guía oficial de Neo4J[9]:

1. Entrar en modo super usuario:

```
Sudo su -s
```

2. Añadir el repositorio:

```
rpm --import https://debian.neo4j.com/neotechnology.gpg.key
sudo nano /etc/yum.repos.d/neo4j.repo
[neo4j]
name=Neo4j RPM Repository
baseurl=https://yum.neo4j.com/stable
enabled=1
gpgcheck=1
(ctrl + X para salir de nano)
```

3. Activar openJDK11 (viene incluido con la instancia, pero sin activar).

```
amazon-linux-extras enable java-openjdk11
```

4. Instalar adaptador de Neo4j:

```
sudo yum install https://dist.neo4j.org/neo4j-java11-
adapter.noarch.rpm
```

5. Instalar Neo4j:

```
yum install neo4j-4.3.2
```

Antes de poner en marcha el servidor hay que hacer unas modificaciones en los ficheros de configuración ya que, por defecto, Neo4J no escucha a todas las direcciones por sus puertos.

El fichero que hay que modificar es /etc/neo4j/neo4j.conf. Las modificaciones que hay que hacer son las siguientes:

- Descomentar la línea dbms.default_lister_address=0.0.0.0.

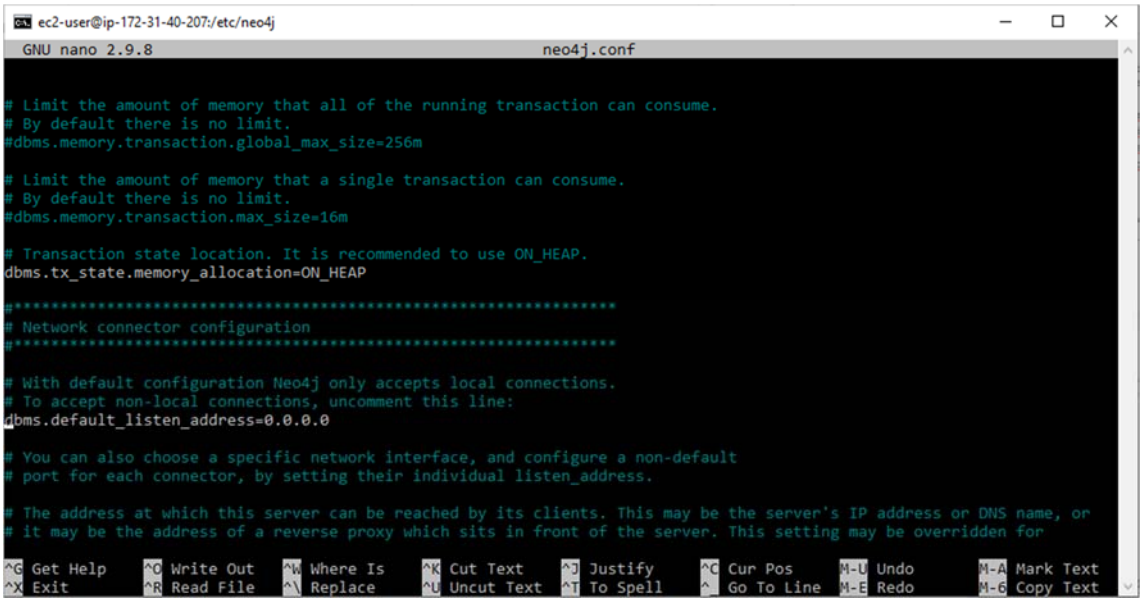

Imagen 20. Modificación del fichero neo4j.conf (I)

- Modificar la configuración de los conectores como se ve en la siguiente imagen:

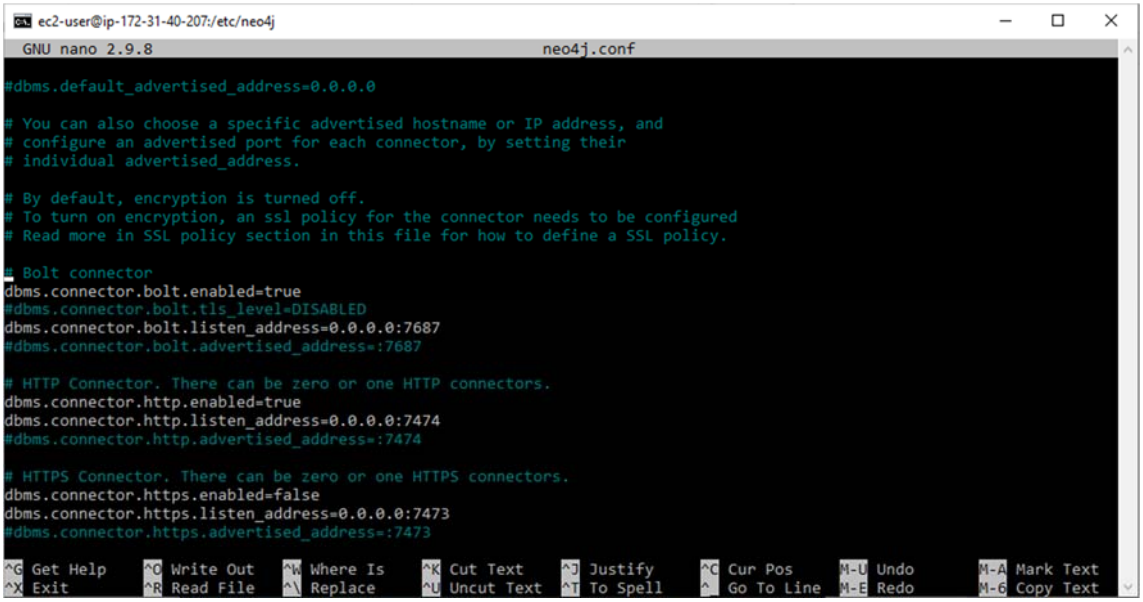

Imagen 21. Modificación del fichero neo4j.conf (II)

- Además, por defecto, la configuración solo trabaja con cien peticiones simultaneas por lo que hay que añadir las siguientes opciones:

```
# Number of Neo4j worker threads.
#dbms.threads.worker_count=

dbms.connector.bolt.thread_pool_min_size=10
dbms.connector.bolt.thread_pool_max_size=1500
dbms.connector.bolt.thread_pool_keep_alive=10m
#*****************************************************************
```

Imagen 22. Modificación del fichero neo4j.conf (III)

Una vez modificado el archivo de configuración ya está todo listo para poder poner en marcha el servidor. Para ello hay que ejecutar el siguiente comando:

```
Sudo /bin/neo4j start
```

Neo4J dispone de una aplicación web para interactuar con la base de datos.

Para acceder a ella hay que poner la dirección IP de la instancia en el buscador y el puerto 7474.

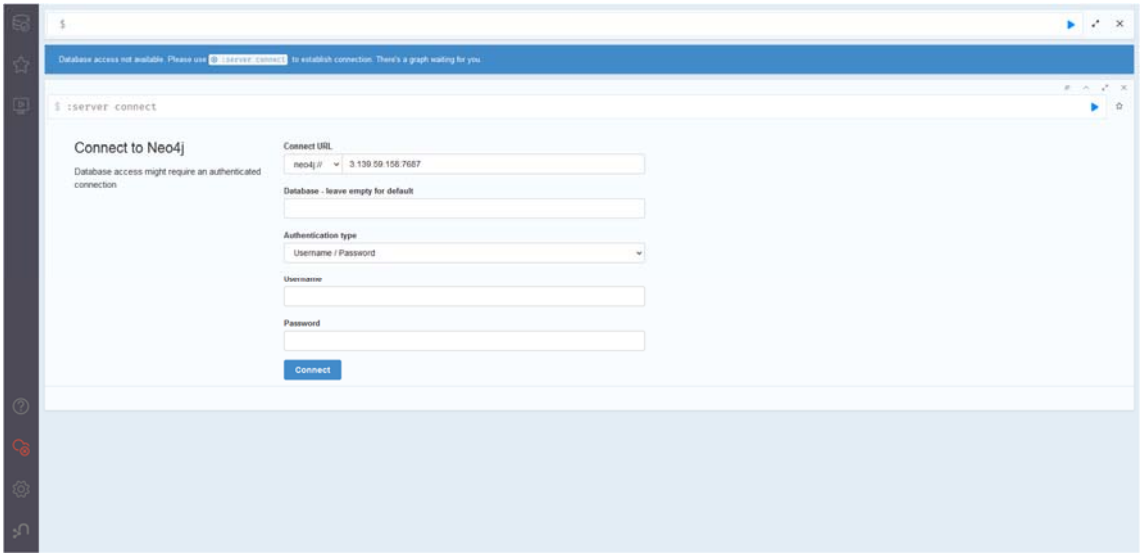

Imagen 23. Inicio de sesión en la aplicación web de Neo4J.

La primera vez que se inicie sesión se debe hacer con el usuario Neo4J y contraseña Neo4J, acto seguido se deberá cambiar la contraseña.

Una vez dentro se puede efectuar cualquier tipo de *query*.

Como ya se mencionó en la sección *2.4. Estructura de Neo4J*, en Neo4J hay que crear los once nodos necesarios para el funcionamiento del sistema.

Para ello hay que efectuar las siguientes consultas:

```
CREATE (:status_node{start: 'false', type: 'read'})
CREATE (:readtest_node{prop1: 'value', prop2: 'value})
CREATE (:readtest_node{prop1: 'value', prop2: 'value})
CREATE (:readtest_node{prop1: 'value', prop2: 'value})
CREATE (:readtest_node{prop1: 'value', prop2: 'value})
```

```
CREATE (:readtest_node{prop1: ‘value’, prop2: ‘value}) 
CREATE (:readtest_node{prop1: ‘value’, prop2: ‘value}) 
CREATE (:readtest_node{prop1: ‘value’, prop2: ‘value}) 
CREATE (:readtest_node{prop1: ‘value’, prop2: ‘value}) 
CREATE (:readtest_node{prop1: ‘value’, prop2: ‘value}) 
CREATE (:readtest_node{prop1: ‘value’, prop2: ‘value})
```

Cabe mencionar que con Neo4J al crear un nodo se le asigna una id única.

Una vez creados todos los nodos necesarios se puede comprobar su creación con la consulta:

```
MATCH (n) RETURN(n)
```

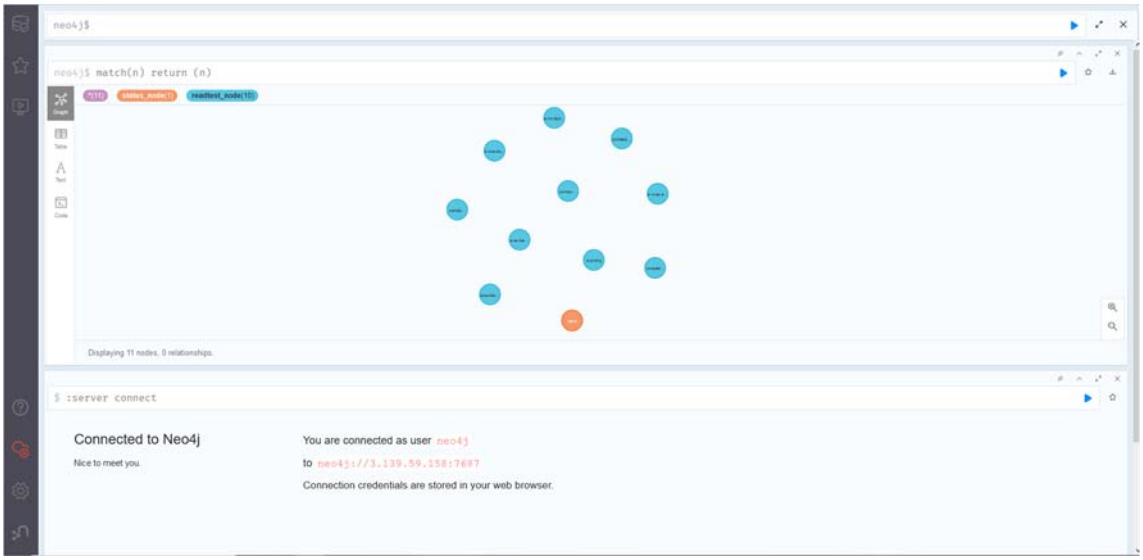

Imagen 24. Resultado de query Match(n) Return(n).

Con esto Neo4J, ya está completamente configurada para las pruebas de estrés.

3.2.7. Configuración de HBase.

Como ya se ha mencionado en secciones anteriores, a la hora de crear la instancia hay que seleccionar la AMI base y el tipo de instancia t2.2xlarge.

Además, hay que habilitar dos puertos de la misma forma que con las anteriores bases de datos. Los puertos son 9090 y 16000.

Para instalar Hbase hay que seguir los siguientes pasos según la guía oficial de Apache Hbase [10]:

(JDK 8 ya está instalado en la AMI base)

1. Descargar el paquete comprimido:

```
Wget https://downloads.apache.org/hbase/2.4.4/hbase-2.4.4-
client-bin.tar.gz
```

2. Descomprimir paquete:

```
Tar xzvf hbase-2.4.4-client-bin.tar.gz
```

Al igual que en Neo4J se han de añadir unas cuantas propiedades en el archivo de configuración. Para ello hay que ejecutar los siguientes comandos:

Cd /hbase-2.3.5/conf

Sudo nano hbase-site.xml

Por defecto, no hay ninguna propiedad en el archivo de configuración, hay que añadir las siguientes:

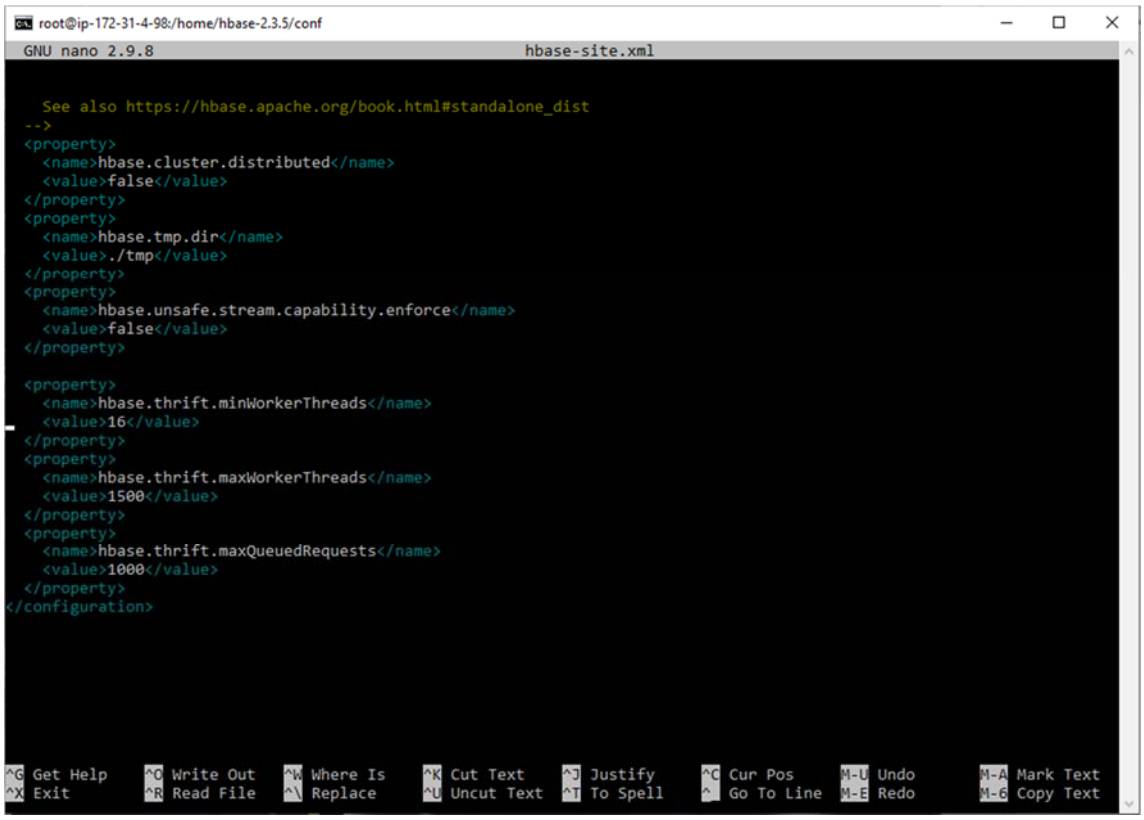

Imagen 25. Propiedades de archivo hbase-site.xml.

Una vez modificado el archivo, ya puede arrancarse el servidor.

Para ello hay que ejecutar los siguientes comandos:

```
/hbase-2.3.5/bin/./start-hbase.sh
```

```
/hbase-2.3.5/bin/hbase thrift start
```

El primero inicia el servidor y el segundo inicia la aplicación a través de la cual el script de Python se comunica con el servidor.

Para entrar en la consola de el servidor hay que ejecutar el siguiente comando:

```
/hbase-2.3.5/bin/hbase start
```

Una vez dentro para crear y configurar las tablas para que funcione la prueba de estrés hay que ejecutar los siguientes comandos:

```
create ‘readtest_table’ , ‘readtest_column’
```

```
create ‘status_table’ , ‘status_column’
```

```
create ‘timeStamps_table’ , ‘timeStamps_column’
```

```
create ‘workers_table’ , ‘workers_column’
```

```
create ‘writetest_table’ , ‘writetest_column’
```

A diferencia que con Casandra, no hay que definir ni las columnas ni el tipo de valor. Lo que se hace es crear las familias de columnas. Estas familias de columnas agrupan las columnas que se pueden agregar, eliminar y modificar de manera flexible.

Para comprobar que se han creado correctamente se puede usar el comando:

```
List
```

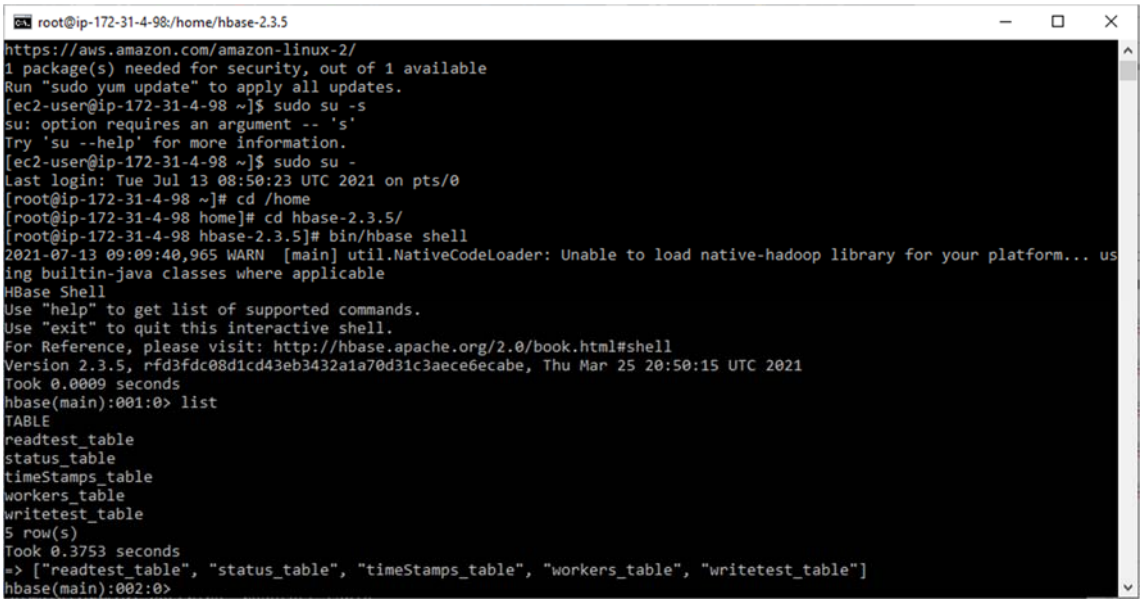

Imagen 26. Comprobación de tablas creadas en Hbase.

Para acabar de configurar la base de datos de cara a la prueba de estrés falta añadir los datos necesarios en la tabla readtest_table y status_table. Para ello hay que ejecutar los siguientes comandos en la consola del servidor:

```
put 'status_table','row1','status_column:start','false'
put 'status_table','row2','status_column:type,'read'
put 'status_table','row2','status_column:type,'read'
put 'readtest_table','row1','readtest_column:col1,'val1'
put 'readtest_table','row2','readtest_column:col2,'val2'
put 'readtest_table','row3','readtest_column:col1,'val1'
put 'readtest_table','row4','readtest_column:col2,'val2'
put 'readtest_table','row5','readtest_column:col1,'val1'
put 'readtest_table','row6','readtest_column:col2,'val2'
put 'readtest_table','row7','readtest_column:col1,'val1'
put 'readtest_table','row8','readtest_column:col2,'val2'
put 'readtest_table','row9','readtest_column:col1,'val1'
put 'readtest_table','row10','readtest_column:col2,'val2'
put 'readtest_table','row11','readtest_column:col1,'val1'
put 'readtest_table','row12','readtest_column:col2,'val2'
put 'readtest_table','row13','readtest_column:col1,'val1'
put 'readtest_table','row14','readtest_column:col2,'val2'
put 'readtest_table','row15','readtest_column:col1,'val1'
put 'readtest_table','row16','readtest_column:col2,'val2'
put 'readtest_table','row17','readtest_column:col1,'val1'
put 'readtest_table','row18','readtest_column:col2,'val2'
```

```
put ’readtest_table’,’row19’,’readtest_column:col1,’val1’
put ’readtest_table’,’row20’,’readtest_column:col2,’val2’
```

Una vez ejecutados estos comandos Hbase está completamente configurada para la prueba de estrés.

3.3 Explicación de los códigos.

Todos los scripts desarrollados para la ejecución de la prueba de estrés siguen la misma estructura.

Antes de comenzar a ver en detalle cada script es relevante revisar unos conceptos para facilitar la compresión de los scripts.

La prueba de estrés consiste en estudiar el tiempo de respuesta de una base de datos cuando un usuario está solicitando o insertando información, cuando cien usuarios solicitan o insertan información y finalmente, cuando mil usuarios solicitan o insertan información.

Inicialmente AWS tiene un límite de sesenta instancias por cuenta. Fue necesario solicitar un aumento de este límite para el proyecto. Aunque el límite se incrementó hasta doscientos usuarios.

Por ello, para lograr simular la cantidad de mil usuarios se ha requerido del uso de *multithreading*, para que cada instancia *worker* simule cinco usuarios cada una.

La librería utilizada se llama *concurrent.futures*. Esta librería permite la ejecución simultanea de tareas aprovechando los procesadores virtuales de las maquinas donde se ejecuta el script.

Dependiendo de la cantidad de procesadores virtuales se pueden ejecutar más o menos tareas simultaneas, en concreto, un total de vCPUs * 5 en el caso de utilizar ThreadPoolExecutor.

Toda la información acerca de la librería de Python utilizada para este propósito se encuentra la API oficial de la librería concurrent.futures [11].

Cabe destacar que la ejecución de estas tareas es asíncrona. Por ello, como se verá a continuación es los scripts, uno de los parámetros de entrada es el tiempo durante el cual un *worker* estará consultando la base de datos (en todas las pruebas se han utilizado cinco segundos), para así garantizar la simultaneidad de las tareas durante un amplio volumen de tiempo.

A continuación, se detallarán los scripts desarrollados.

3.3.1 Explicación de los códigos de MongoDB.

stworkerMONGODB.py

```
from pymongo import MongoClient
from datetime import datetime,timedelta
import argparse
from requests import get
from statistics import mean
import os
from concurrent.futures import ThreadPoolExecutor
from random import randrange
```

Esta parte corresponde a la importación de todas las librerías usadas.

```
parser = argparse.ArgumentParser()
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance. ")
parser.add_argument("-s", dest="seconds", help="introduce the live
time in seconds. ")
parser.add_argument("-t", dest="nThreads",default =0, help="(optional)
introduce the number of threads in the pool 1 to 4. If is not set the
threadpool executor won't be used. ")
```

Esta parte es común a todos los scripts de tipo *worker*.

Se añade la funcionalidad de pasar como argumentos a la hora de ejecutar el script; la IP del servidor donde se encuentra la base de datos, el tiempo que el *worker* en cuestión estará estresando la base de datos y el número de *threads* que usará.

Este último argumento es el que determinará cuantos usuarios simulará cada instancia.

```
publicIpServer = parser.parse_args().ip
liveTime = int(parser.parse_args().seconds)
nThreads = int(parser.parse_args().nThreads)

myPublicIp = get('https://api.ipify.org').text

newClient=MongoClient(publicIpServer,27017)
```

Se declaran las variables globales.

publicIpServer: recoge el valor pasado como argumento correspondiente a la dirección IP de la base de datos a analizar.

liveTime: recoge el valor pasado como argumento correspondiente al tiempo que se estará estresando la base de datos por parte del usuario.

nThreads: recoge el valor pasado como argumento correspondiente al número de usuarios que se simularan.

myPublicIP: mediante la función *get* de la librería *request*, hace una conexión http a una página que devuelve la IP pública de la maquina desde donde se ejecuta.

newClient: mediante la función MongoClient de la librería *pymongo* se genera un conector a la base de datos pasando como argumento la IP de la base de datos a analizar y el puerto por donde escucha.

A partir de aquí, se definen dos funciones principales que básicamente hacen lo mismo, la única diferencia es que una está diseñada para el caso en el que sólo se simula un usuario y la otra para cuando se simulan más.

```
def mainFunction():


      newClient.status.workers.insert_one({"workerIp":str(myPublicIp)
      ,"status":"working"})

      print("Waiting for master...")

      listenFunction(newClient)

      print("Next step...")
      stressTestTypeCursor =
newClient.status.status_collection.find({"statusObjectName":"type"},
{"status":1})
      stressTestTypeCursorList= list(stressTestTypeCursor)
      stressTestType = stressTestTypeCursorList[0].get('status')
      timeStart = datetime.now()
      timeStampsList =[]

      while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):
            if stressTestType == "read":
                timeStampsList.append(readFunction(newClient))
            # elif stressTestType =="write":
            else :
                a=0
                queryList = []
                while a < 10:

      queryList.append({"id":str(myPublicIp)+str(randrange(0,10000000
      0,1)),"resultado":"2-6"})
                    a=a+1

      timeStampsList.append(writeFunction(newClient,queryList))

      try:
```

```
            newClient.status.timeStamps.insert_one({"ip":str(myPublic
      Ip),"nQuery":str(len(timeStampsList)),"avgQuery":
      mean(timeStampsList)})

       except Exception as e:
             print(e)
             pass


            newClient.status.workers.update_one({"workerIp":str(myPub
            licIp)},{"$set":{"status":"completed"}})
        newClient.close()

      print(str(len(timeStampsList)))
      print(str(mean(timeStampsList)))
```

Utilizando el conector creado anteriormente se envía una *query* de escritura que inserta un documento en la base de datos status, en la colección *workers*, con los campos workerIP cuyo valor es la variable global myPublicIP y status cuyo valor es *working*.

Después se imprime por pantalla el mensaje “Waiting for master”.

Y se llama a la función listenFunction pasándole como argumento el conector a la base de datos.

Al finalizar la función, explicada más adelante, se hace una *query* de lectura a la base de datos status, colección status, obteniendo el valor del campo status donde el campo statusObjectName tiene el valor de *type*. Este valor leído es *read* o *write* dependiendo de la prueba que vaya a ejecutarse.

Las dos siguientes líneas son necesarias para transformar el tipo de la información leída de la base de datos a un str ya que al ejecutar el método *find* se recibe un objeto de tipo mongo cursor.

Después se guarda en la variable timeStart el valor del tiempo a través de la función datetime.now() de la librería *datetime*.

Después se declara una lista vacía timeStampList.

Después se entra en un bucle en el que en cada iteración se consulta el valor del tiempo actual y si es mas pequeño que el valor guardado anteriormente más el valor de la variable global liveTime sigue iterando. De esta forma este bucle dura aproximadamente el tiempo establecido por esta variable global.

Este bucle dependiendo del valor de la variable obtenida anteriormente stressTestType, llamará a dos funciones definidas posteriormente, con las cuales se estresa la base de datos con consultas de escritura o lectura. Estas

dos funciones retornan el valor que a tomado en procesarse la *query* efectuada y se añade a la lista inicialmente vacía timeStampsList.

En el caso de la prueba de tipo *write*, se prepara una *query* que insertará diez documentos. En MongoDB no puede haber duplicidad de datos en dos documentos, por lo que en el campo id se añade la variable global myPublicIp junto con un valor numérico comprendido uno y diez millones.

Transcurrido el bucle, se inserta un documento en la base de datos status, en la colección timeStamps, un documento que contiene la variable global myPublicIp, el número de *queries* que ha podido realizar durante el bucle, y la media de los valores de procesamiento de todas las *queries*. (Omitir el try/except ya que se ha usado para el debugging).

Finalmente se actualiza el documento de la base de datos status, colección *workers* insertado con anterioridad, cambiando el valor del campo status de *working* a *completed* y se muestra por pantalla la longitud y la media de la lista timeStampsList.

```
def mainFunction2(threadID):

      newClient.status.workers.insert_one({"workerIp":str(myPublicIp)
      ,"threadID": str(threadID),"status":"working"})

      print("Waiting for master...")

      listenFunction(newClient)

      print("Next step... ThreadID: "+str(threadID))
      stressTestTypeCursor =
newClient.status.status_collection.find({"statusObjectName":"type"},
{"status":1})
      stressTestTypeCursorList= list(stressTestTypeCursor)
      stressTestType = stressTestTypeCursorList[0].get('status')
      timeStart = datetime.now()
      timeStampsList =[]

      while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):

            if stressTestType == "read":
                timeStampsList.append(readFunction(newClient))
            # elif stressTestType =="write":
            else :
                # print(queryList)
                a=0
                queryList = []
                while a < 10:

                  queryList.append({"id":str(myPublicIp)+str(threadID
                  )+str(randrange(0,100000000,1)),"resultado":"2-6"})
                    a=a+1

      timeStampsList.append(writeFunction(newClient,queryList))
```

```
        try:

              newClient.status.timeStamps.insert_one({"ip(threadID)":st
              r(myPublicIp)+"("+str(threadID)+")","nQuery":str(len(time
              StampsList)),"avgQuery": mean(timeStampsList)})
        except Exception as e:
              print(e)
              pass


        newClient.status.workers.update_one({"workerIp":str(myPublicIp)
        ,"threadID":str(threadID)},{"$set":{"status":"completed"}})
          newClient.close()

        print(str(len(timeStampsList)))
        print(str(mean(timeStampsList)))
```

La función mainFunction2 es la equivalente a la anterior pero adaptada para el caso de simular más de un usuario.

Los cambios son añadir el valor de la variable threadID, pasado como argumento a la hora de llamar a la función, inmediatamente después de todos los llamamientos a la variable myPublicIP. Esto se ha hecho para evitar los problemas de duplicidad de datos al insertar nuevos documentos.

```
def listenFunction(client):

    status = "false"
    while status == "false":
        statusCursor =
client.status.status_collection.find({"statusObjectName":"start"},{"
status":1})
        statusCursorList= list(statusCursor)
        status = statusCursorList[0].get('status')
```

La función listerFunction recibe como argumento el conector a la base de datos.

Lo que hace es mediante un bucle, consultar por el valor de la base de datos status, colección status_collection, campo status, donde el valor del campo statusObjectName es *start*.
Este valor inicialmente siempre vale *false* tal y como se ha explicado en capítulos anteriores. La instancia master mediante el script stmasterMONGODB.py cambia este valor a *true* y es entonces cuando se finaliza esta función y se sigue ejecutando del resto del código.

```
def readFunction(client):

      timeStamp0=datetime.now()
```

```
        newClient.test.readtest.aggregate([{"$group":{"_id":"null","avg
        ":{"$sum":1}}}])
        return((datetime.now()-timeStamp0).total_seconds())
```

La función readFunction, recibe como argumento el conector a la base de datos.

Se guarda el instante temporal timeStamp0, después se efectúa la *query* a la base de datos test, colección readtest donde básicamente se cuentan los documentos que hay (siempre en todas las pruebas de todas las bases de datos son diez).

Finalmente devuelve el resultado de restar el instante temporal en ese momento menos el guardado anteriormente. De esta forma se calcula cuánto se ha tardado en procesar la query.

```
def writeFunction(client,queryList):

    try:
        timeStamp0=datetime.now()
        newClient.test.writetest.insert_many(queryList)
        return((datetime.now()-timeStamp0).total_seconds())

    except Exception as e:
        print("\n"+str(e)+"\n")
        pass
```

La función writeFunction recibe como argumento el conector a la base de datos y la lista con las *queries* a realizar.

Al igual que readFunction guarda un instante temporal justo antes de relizar la *query* que inserta diez filas a la base de datos test, colección writetest.

Finalmente devuelve al igual que en readFunction, el valor de procesamiento de la *query* de escritura.

(Omitir el try/except ya que se usó para el *debugging*)

```
maxThread = os.cpu_count()*5

if maxThread < nThreads and nThreads > 0:

    print("You have exceed threads limit, it has been set to
"+str(maxThread)+".")
    nThreads = maxThread

if nThreads > 0:

    mainFunctionArgs = list(range(0,nThreads))
```

```
    with ThreadPoolExecutor() as executor:
        results = executor.map(mainFunction2, mainFunctionArgs)

else:
    mainFunction()
```

Finalmente, esta parte del código primero calcula cual es la cantidad máxima de *threads* que se pueden usar desde la máquina que se va a ejecutar el script.

Si se ha pasado como argumentos más de la cantidad permitida se comunica por pantalla y se cambia el valor de la variable global nThreads a el valor de la variable recién calculada maxTread.

Después si se llama a la función principal dependiendo de si se ha elegido simulas un usuario o más.

En este último caso se hace uso de multithreading para la simulación de más de un usuario.

ThreadPoolExecutor se encarga de enviar las tareas a distintos hilos lógicos del procesador mediante la función *map*.

Los argumentos a pasar son la función a realizar por el *thread* y los argumentos que se le pasan a dicha función en formato de lista. De manera que cada valor de la lista, en este caso una cuenta desde 0 hasta el número de *threads* a usar, corresponde al valor del argumento ThreadID de la función mainFunction2(threadID).

stmasterMONGODB.py

```
from pymongo import MongoClient
from datetime import datetime,timedelta
import time
import argparse
from statistics import mean
```

Esta parte corresponde a la importación de todas las librerías usadas.

```
parser = argparse.ArgumentParser()
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance")
parser.add_argument("-t", dest="type",default="read",help= "introduce
what type of stress test to perform: read or write")

publicIpServer = parser.parse_args().ip
stressTestType = parser.parse_args().type
```

Al igual que en el script *worker*, en esta parte se definen las variables globales que reciben su valor a través de los argumentos al ejecutar el script.

La variable publicIpServer corresponde a la dirección IP del servidor de la base de datos a analizar.

La variable stressTestTyple corresponde a la prueba que va a realizarse.

```
def startFunction():

      print("Starting...")
      newClient=MongoClient(publicIpServer,27017)

      newClient.status.status_collection.update_one({"statusObjectNam
      e":"type"},{"$set":{"status":str(stressTestType)}})

      newClient.status.status_collection.update_one({"statusObjectNam
      e":"start"},{"$set":{"status":"true"}})

      nWorkers=
newClient.status.workers.aggregate([{"$group":{"_id":"null","avg":{"
$sum":1}}}])
      nWorkers_list=list(nWorkers)
      print(str(nWorkers_list[0]['avg'])+" users will participate.")
      print("Stress Test type: " + str(stressTestType))
```

La función startFunction es la encargada de dar comienzo a la prueba.

Primero imprime por consola el mensaje “Starting…”.

Crea un conector a la base de datos.

Actualiza la base de datos status, colección status_collection, campo status donde el campo statusObjectName tiene valor *type*, con el valor correspondiente a la variable glogal stressTestType.

Actualiza la base de datos status, colección status_collection, campo status donde el campo statusObjectName tiene el valor *start*, con el valor “true”.

A partir de este momento, el script *worker* que estaría en el bucle de la función listenFunction, leerían el “true” y saldría del bucle prosiguiendo con la ejecución del script.

Crea la variable nWorkers con el valor obtenido de hacer una *query* que cuenta cuantos documentos existen en la colección *workers* de la base de datos status.

Después se coge el valor de tipo *string* del objeto mongo cursor resultado de la prueba.

Se muestra por pantalla el número de *workers* leído y el tipo de prueba que se va a ejecutar.

```
def measureFunction():

      newClient = MongoClient(publicIpServer,27017)
      print("Measuring...")
      end = "false"
      while end == "false":

        cursor = newClient.status.workers.distinct("status")
        cursorList= list(cursor)
        if len(cursorList) == 1 and cursorList[0] == "completed":
            end = "true"

      Cursor=
newClient.status.timeStamps.aggregate([{"$group":{"_id":"null","avg"
:{"$avg":"$avgQuery"}}}])
      CursorList= list(Cursor)
      timeAVG = CursorList[0]["avg"]
      print(timeAVG)


      newClient.status.status_collection.update_one({"statusObjectNam
      e":"start"},{"$set":{"status":"false"}})
    newClient.status.workers.delete_many({})
    newClient.status.timeStamps.delete_many({})
    newClient.test.writetest.delete_many({})
```

La función measureFunction se encarga de resetear las colecciones para que se pueda volver a efectuar una prueba en el futuro, además de mostrar por pantalla el resultado final.

Primero crea un conector a la base de datos.

Muestra por pantalla el mensaje “Measuring…”

Comienza un bucle en el que constantemente comprueba que todos los documentos de la base de datos status, colección workers, tengan el valor en el campo status *completed*. Cuando esto ocurre sale del bucle.

Mediante una *query* de lectura obtiene la media de los valores de la base de datos status, colección timeStamps, campo avgQuery. Estos datos corresponden a las medias registradas de los resultados calculados por los *workers*.

Se muestra por pantalla el resultado.
Finalmente, actualiza en la base de datos status, colección status_collection, campo status donde el campo statusObjectName vale *start*, con el valor “false”. De este modo si se inicia otra prueba en un futuro los *workers* se quedarán a la espera nuevamente del master en la función listenFunction.

También se limpia de datos las colecciones workers, timeStamps y writetest.

```
startFunction()
measureFunction()
```

Se llama a las dos funciones en orden.

3.3.2 Explicación de los códigos de Cassandra.

stworkerCASSANDRA.py

```
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
from datetime import datetime,timedelta
import argparse
from requests import get
from statistics import mean
import os
from concurrent.futures import ThreadPoolExecutor


parser = argparse.ArgumentParser()
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance. ")
parser.add_argument("-s", dest="seconds", help="introduce the live
time in seconds. ")
parser.add_argument("-t", dest="nThreads",default =0, help="(optional)

publicIpServer = parser.parse_args().ip
liveTime = int(parser.parse_args().seconds)
nThreads = int(parser.parse_args().nThreads)

myPublicIp = get('https://api.ipify.org').text
```

Esta parte del código es exactamente similar en todas las bases de datos, a diferencia de que cada una importa su librería correspondiente para crear los conectores a la base de datos.

```
while True:
   try:
       auth_provider =
PlainTextAuthProvider(username='cassandra',password='cassandra')
       cluster = Cluster(contact_points=[publicIpServer],
port=9042,auth_provider=auth_provider)
       session = cluster.connect()
       break
   except Exception as e:
       print(e)
```

Se crea el conector con la base de datos.

Durante la fase de *debugging* del código surgieron varios problemas a la hora de crear los conectores con Cassandra. Por eso es necesario el uso de este bucle que básicamente intenta crear la conexión y si no lo logra lo vuelve a intentar hasta conseguirlo.

A diferencia de MongoDB, además de especificar el puerto y la dirección IP de la base de datos hay que indicar un nombre de usuario y contraseña. El mismo que se usaría para acceder a la consola del servidor.

```
def mainFunction():

        query = "begin batch "

        for i in list(range(0,10)):
            query = query + "insert into stresstest.writetest (id,
value) values ('id"+str(myPublicIp)+str(i)+"','value"+str(i)+"');"

        session.execute("insert into stresstest.workers (workerIP,
status) values ('"+str(myPublicIp)+"','working')")

        print("Waiting for master...")

        listenFunction(session)

        print("Next step...")
        result = session.execute("select status from
stresstest.status_table where statusObjectName ='type'")
        stressTestType = result[0].status
        timeStart = datetime.now()
        timeStampsList =[]

        while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):
            if stressTestType == "read":
                timeStampsList.append(readFunction(session))
            # elif stressTestType =="write":
            else :
                timeStampsList.append(writeFunction(session,query))


        session.execute("insert into stresstest.timeStamps (ip,
nQuery, avgQuery) values
('"+str(myPublicIp)+"',"+str(len(timeStampsList))+","+str(mean(timeSta
mpsList))+");")
        session.execute("update stresstest.workers set status =
'completed' where workerIP ='"+str(myPublicIp)+"'")

        print(str(len(timeStampsList)))
        print(str(mean(timeStampsList)))

def mainFunction2(threadID):

        query = "begin batch "

        for i in list(range(0,10)):
            query = query + "insert into stresstest.writetest (id,
value) values
('"+str(myPublicIp)+str(threadID)+str(i)+"','value"+str(i)+"');"

        while True:
            try:
                session.execute("insert into stresstest.workers
(workerIP,status) values
('"+str(myPublicIp)+"("+str(threadID)+")'"+",'working')")
                break
            except Exception as e:
                print(e)

        print("Waiting for master...")
```

```
        listenFunction(session)

        print("Next step... ThreadID: "+str(threadID))
        result = session.execute("select status from
stresstest.status_table where statusObjectName ='type'")
        stressTestType = result[0].status
        timeStart = datetime.now()
        timeStampsList =[]

        while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):

            if stressTestType == "read":

                timeStampsList.append(readFunction(session))
            # elif stressTestType =="write":
            else :
                timeStampsList.append(writeFunction(session,query))

        try:
            session.execute("insert into stresstest.timeStamps (ip,
nQuery, avgQuery) values
('"+str(myPublicIp)+"("+str(threadID)+")',"+str(len(timeStampsList))+"
,"+str(mean(timeStampsList))+");")
            session.execute("update stresstest.workers set status =
'completed' where workerIP
='"+str(myPublicIp)+"("+str(threadID)+")';")
        except Exception as e:
            print(e)

        print(str(len(timeStampsList)))
        print(str(mean(timeStampsList)))
```

Las funciones mainFunction y mainFunction2 actúan de manera similar que con MongoDB.

Las diferencias están en que el conector no tiene funciones especificas para tratar los datos, sino que ejecuta un comando como si se estuviese en la consola del servidor.

Por ello en todas las *queries* se utiliza la sintaxis de Cassandra la cual es muy similar a la sintaxis SQL.

En algunos puntos de ambas funciones se ven varios bucles similares al de la creación del conector. Pero realmente no son necesarios pues el problema no residía aquí.

Otra consideración que se ha tenido en cuenta a la hora de insertar información es que, al tratarse de una base de datos de modelo Key-Value, los valores de la columna definida en las secciones anteriores como clave primaria no esté duplicada.

```
def listenFunction(session):
```

```
    status = "false"
    while status == "false":
        result = session.execute("select status from
stresstest.status_table where statusObjectName ='start'")
        status = result[0].status

def readFunction(session):

    timeStamp0=datetime.now()
    session.execute("select count(id) from stresstest.readtest")
    return((datetime.now()-timeStamp0).total_seconds())

def writeFunction(session,query):

    try:

        timeStamp0=datetime.now()
        session.execute(query+" apply batch")
        return((datetime.now()-timeStamp0).total_seconds())

    except Exception as e:
        print("\n"+str(e)+"\n")
        pass
```

Las funciones listerFunction, readFunction y writeFunction sólo cambian, nuevamente, en la sintaxis a la hora de ejecutar las queries.

```
maxThread = os.cpu_count()*5

if maxThread < nThreads and nThreads > 0:

    print("You have exceed threads limit, it has been set to
"+str(maxThread)+".")
    nThreads = maxThread

if nThreads > 0:

    mainFunctionArgs = list(range(0,nThreads))
    with ThreadPoolExecutor() as executor:
        results = executor.map(mainFunction2, mainFunctionArgs)

else:
    mainFunction()
```

El final del código es exactamente igual para todas las bases de datos.

stmasterCASSANDRA.py

```
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
from datetime import datetime,timedelta
import time
import argparse
from statistics import mean
import numpy as np


parser = argparse.ArgumentParser()
```

```
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance")
parser.add_argument("-t", dest="type",default="read",help= "introduce
what type of stress test to perform: read or write")

publicIpServer = parser.parse_args().ip
stressTestType = parser.parse_args().type

def startFunction():

    print("Starting...")
    auth_provider =
PlainTextAuthProvider(username='cassandra',password='cassandra')
    cluster = Cluster(contact_points=[publicIpServer],
port=9042,auth_provider=auth_provider)
    session = cluster.connect()
    session.execute("update stresstest.status_table set status =
'"+str(stressTestType)+"' where statusObjectName = 'type'")
    session.execute("update stresstest.status_table set status =
'true' where statusObjectName = 'start'")

    nWorkers = session.execute("select count(workerIP) as nworkers
from stresstest.workers")
    nWorkers_list=list(nWorkers)

    print(str(nWorkers_list[0].nworkers)+" users will participate.")
    print("Stress Test type: " + str(stressTestType))

def measureFunction():

    auth_provider =
PlainTextAuthProvider(username='cassandra',password='cassandra')
    cluster = Cluster(contact_points=[publicIpServer],
port=9042,auth_provider=auth_provider)
    session = cluster.connect()

    print("Measuring...")
    end = "false"
    while end == "false":

        workers = session.execute("select status from
stresstest.workers")
        workers_list = list(workers)
        if len(np.unique(workers_list)) == 1 and
np.unique(workers_list)[0] == "completed":
            end = "true"

    avg = session.execute("select avg(avgQuery) from
stresstest.timeStamps")
    avg_list = list(avg)
    print(avg_list[0].system_avg_avgquery)

    session.execute("update stresstest.status_table set status =
'false' where statusObjectName = 'start'")
    session.execute("truncate stresstest.workers")
    session.execute("truncate stresstest.timeStamps")
    session.execute("truncate stresstest.writeTestTable")


startFunction()
measureFunction()
```

El código master a excepción de la sintaxis a la hora de efectuar las *queries* y la parte resaltada, es exactamente igual a la versión del código diseñado para MongoDB.

La parte resaltada es la encargada de comprobar que no queda ningún *worker* en estado *working*. Una limitación que tiene Cassandra es, la falta de algunos agregados en la sintaxis. En este caso el distinct().

Lo que se hace es guardar todos los valores de los estados en una lista y con la librería *numpy* se obtienen los distintos valores y se hace la comprobación deseada.

3.3.3 Explicación de los códigos de Neo4j.

stworkerNEO4J.py

```
from neo4j import GraphDatabase
from datetime import datetime,timedelta
import argparse
from requests import get
from statistics import mean
import os
from concurrent.futures import ThreadPoolExecutor


parser = argparse.ArgumentParser()
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance. ")
parser.add_argument("-s", dest="seconds", help="introduce the live
time in seconds. ")
parser.add_argument("-t", dest="nThreads",default =0, help="(optional)
introduce the number of threads in the pool 1 to 4. If is not set the
threadpool executor won't be used. ")


publicIpServer = parser.parse_args().ip
liveTime = int(parser.parse_args().seconds)
nThreads = int(parser.parse_args().nThreads)
# publicIpServer = "18.223.151.148"
# liveTime = 5
# nThreads = 0

myPublicIp = get('https://api.ipify.org').text
```

Esta parte es igual en todos los scripts.

```
url = "neo4j://"+str(publicIpServer)+":7687"
driver = GraphDatabase.driver(url, auth=("neo4j", "admin"))
```

Al igual que en Cassandra, el conector requiere de un usuario y contraseña.

```
def mainFunction():


        query = ""
        for i in range(0,10):
            query = query + " CREATE (:writetest_node
{id:'"+str(myPublicIp)+"',value:'randomvalue'})"

        driver.session().run("CREATE
(n:worker_node{workerIp:'"+str(myPublicIp)+"',status:'working'})")

        print("Waiting for master...")

        listenFunction(driver)

        print("Next step...")
        result = driver.session().run("match(n:status_node) return
(n.type) as type")
        stressTestType = result.data()[0]['type']
        timeStart = datetime.now()
```

```
        timeStampsList =[]

        while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):
            if stressTestType == "read":
                timeStampsList.append(readFunction())
            # elif stressTestType =="write":
            else :
                timeStampsList.append(writeFunction(query))

        driver.session().run("CREATE
(n:timestamp_node{ip:'"+str(myPublicIp)+"',nQuery:"+str(len(timeStamps
List))+",avgQuery:"+str(mean(timeStampsList))+"})")
        driver.session().run("MATCH (n:worker_node) WHERE
n.workerIp='"+str(myPublicIp)+"' SET n.status ='completed'")

        print(str(len(timeStampsList)))
        print(str(mean(timeStampsList)))

def mainFunction2(threadID):

        query = ""
        for i in range(0,10):
            query = query + " CREATE (:writetest_node
{id:'"+str(myPublicIp)+"("+str(threadID)+"',value:'randomvalue'})"


        driver.session().run("CREATE
(:worker_node{workerIp:'"+str(myPublicIp)+"("+str(threadID)+")',status
:'working'})")

        print("Waiting for master...")

        listenFunction(driver)

        print("Next step... ThreadID: "+str(threadID))
        result = driver.session().run("match(n:status_node) return
(n.type) as type")
        stressTestType = result.data()[0]['type']
        timeStart = datetime.now()
        timeStampsList =[]

        while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):

            if stressTestType == "read":

                timeStampsList.append(readFunction())
            # elif stressTestType =="write":
            else :
                timeStampsList.append(writeFunction(query))

        try:
            driver.session().run("CREATE
(n:timestamp_node{ip:'"+str(myPublicIp)+"("+str(threadID)+"',nQuery:"+
str(len(timeStampsList))+",avgQuery:"+str(mean(timeStampsList))+"})")
            driver.session().run("MATCH (n:worker_node) WHERE
n.workerIp='"+str(myPublicIp)+"("+str(threadID)+")' SET n.status
='completed'")
        except Exception as e:
            print(e)
```

```
        print(str(len(timeStampsList)))
        print(str(mean(timeStampsList)))
```

En las funciones mainFunction y mainFunction2 no se ha requerido de ningún bucle de repetición en caso de fallo.

La estructura es la misma que en los scripts adaptados a las otras bases de datos.

Al igual que Cassandra, el conector no dispone de funciones específicas, sino que ejecuta comandos de consola como si se estuviera en el servidor de la base de datos.

```
def listenFunction(driver):

    status = "false"
    while status == "false":
        result = driver.session().run("match(n:status_node) return
(n.start) as start")
        status = result.data()[0]['start']
```

La función listenFunction se comporta igual que en los scripts para las otras bases.

```
def readFunction():



    while True:
        try:
            driver2 = GraphDatabase.driver(url, auth=("neo4j",
"admin"))
            timeStamp0=datetime.now()
            driver2.session().run("match(n:readtest_node) return (n)")
            driver2.close()
            timeStamp1=datetime.now()
            break
        except Exception as e:
            print (e)

    return((timeStamp1-timeStamp0).total_seconds())

def writeFunction(query):

    try:

        driver2 = GraphDatabase.driver(url, auth=("neo4j", "admin"))
        timeStamp0=datetime.now()
        driver2.session().run(query)
        timeStamp1=datetime.now()
        driver2.close()
        return((timeStamp1-timeStamp0).total_seconds())

    except Exception as e:
        print("\n"+str(e)+"\n")
```

```
        pass
```

A diferencia que, en los scripts para las anteriores bases de datos, el conector no era capaz de procesar las *queries* al simular varios usuarios.

Para solventar este problema en las funciones readFunction y writeFunction se creaban nuevos conectores para cada *query* y se cerraba al acabar.

Aun así, las marcas temporales usadas para calcular la *query* sólo miden el tiempo de procesamiento de la *query* en sí y no de la creación del nuevo conector.

```
maxThread = os.cpu_count()*5

if maxThread < nThreads and nThreads > 0:

    print("You have exceed threads limit, it has been set to
"+str(maxThread)+".")
    nThreads = maxThread

if nThreads > 0:

    mainFunctionArgs = list(range(0,nThreads))
    with ThreadPoolExecutor() as executor:
        results = executor.map(mainFunction2, mainFunctionArgs)

else:
    mainFunction()
```

El final del código es exactamente igual para todas las bases de datos.

stmasterNEO4J.py

```
from neo4j import GraphDatabase
from datetime import datetime,timedelta
import time
import argparse
from statistics import mean
import numpy as np


parser = argparse.ArgumentParser()
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance")
parser.add_argument("-t", dest="type",default="read",help= "introduce
what type of stress test to perform: read or write")

publicIpServer = parser.parse_args().ip
stressTestType = parser.parse_args().type
# stressTestType = "read"
# publicIpServer = "18.223.151.148"

url = "neo4j://"+str(publicIpServer)+":7687"

def startFunction():

    print("Starting...")
```

```
    driver = GraphDatabase.driver(url, auth=("neo4j", "admin"))
    driver.session().run("MATCH(n:status_node) SET n.type =
'"+str(stressTestType)+"'")
    driver.session().run("MATCH(n:status_node) SET n.start = 'true'")

    result = driver.session().run("MATCH(n:worker_node) RETURN
COUNT(n) as count")
    nWorkers = result.data()[0]['count']

    print(str(nWorkers)+" users will participate.")
    print("Stress Test type: " + str(stressTestType))

def measureFunction():

    driver = GraphDatabase.driver(url, auth=("neo4j", "admin"))
    print("Measuring...")
    end = "false"
    while end == "false":

        result = driver.session().run("MATCH (n:worker_node) WITH
DISTINCT (n.status) as status RETURN status")
        result_list = result.data() ## una vez que se ejecuta el
result.data() se borra el objeto. Por ello hay que guardarlo en una
lista si queremos usarlo varias veces.
        if len(result_list) == 1  and result_list[0]['status'] ==
"completed":
            end = "true"

    avg = driver.session().run("MATCH (n:timestamp_node) RETURN
AVG(n.avgQuery) as avg")
    print(avg.data()[0]['avg'])

    driver.session().run("MATCH(n:status_node) SET n.start = 'false'")
    driver.session().run("MATCH(n:worker_node) DELETE(n)")
    driver.session().run("MATCH(n:timestamp_node) DELETE(n)")
    driver.session().run("MATCH(n:writetest_node) DELETE(n)")


startFunction()
measureFunction()
```

El script master actúa de la misma manera que los anteriores.

Los únicos cambios realizados son los correspondientes a la sintaxis.

En este caso, la agregación Distinct() sí está incluida en la sintaxis de Neo4J por tanto no ha sido necesaria la utilización de la librería *numpy* como ocurría con Cassandra.

Un aspecto a destacar es, que la información recibida al realizar queries de lectura se guarda en un objeto de tipo result y cuando se hace uso de la información esta desaparece haciéndola inaccesible.

No fue hasta el punto resaltado cuando este detalle afectó al código. Ya que surgió un problema cada vez que se ejecutaba la función len(result_list),

después cuando se quería acceder al dato “status” no había información y saltaba una excepción de índices en la lista.

El problema se solventó guardando la información previamente en la lista result_list.

3.3.4 Explicación de los códigos de HBase.

stworkerHBASE.py

```
import happybase
from datetime import datetime,timedelta
import argparse
from requests import get
from statistics import mean
import os
from concurrent.futures import ThreadPoolExecutor
from random import randrange


parser = argparse.ArgumentParser()
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance. ")
parser.add_argument("-s", dest="seconds", help="introduce the live
time in seconds. ")
parser.add_argument("-t", dest="nThreads",default =0, help="(optional)
introduce the number of threads in the pool 1 to 4. If is not set the
threadpool executor won't be used. ")


publicIpServer = parser.parse_args().ip
liveTime = int(parser.parse_args().seconds)
nThreads = int(parser.parse_args().nThreads)
# publicIpServer = "3.136.19.111"
# liveTime = 5
# nThreads = 0

myPublicIp = get('https://api.ipify.org').text
```

Esta parte es común a todos los scripts.

```
def mainFunction():

        connection = happybase.Connection(publicIpServer)

        table = connection.table('writetest_table')
        b = table.batch()
        for i in range(0,10):
            b.put('status_row_'+str(randrange(1,100000000)),
{'writetest_column:col1': 'val1','writetest_column:col2': 'val2'})

        while True:
            try:
                workers_table = connection.table('workers_table')
                workers_table.put(str(myPublicIp),
{'workers_column:status': 'working'})
                break
            except Exception as e:
                print(e)

        print("Waiting for master...")

        listenFunction(connection)

        print("Next step...")
```

```
        while True:
            try:
                status_table = connection.table('status_table')
                status_column = status_table.row('status_row')

stressTestType=status_column[b'status_column:type'].decode('ascii')
                break
            except Exception as e:
                print(e)

        timeStart = datetime.now()
        timeStampsList =[]

        while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):
            if stressTestType == "read":
                timeStampsList.append(readFunction(connection))
            # elif stressTestType =="write":
            else :
                timeStampsList.append(writeFunction(connection,b))

        while True:
            try:
                timeStamps_table =
connection.table('timeStamps_table')
                timeStamps_table.put(str(myPublicIp),
{'timeStamps_column:nQuery':
str(len(timeStampsList)),'timeStamps_column:avgQuery':str(mean(timeSta
mpsList))})
                workers_table.put(str(myPublicIp),
{'workers_column:status': 'completed'})
                break
            except Exception as e:
                print(e)

        print('\n'+str(len(timeStampsList)))
        print('\n'+str(mean(timeStampsList)))

def mainFunction2(threadID):

        connection = happybase.Connection(publicIpServer)

        table = connection.table('writetest_table')
        b = table.batch()
        for i in range(0,10):
            b.put('status_row_'+str(randrange(1,100000000)),
{'writetest_column:col1': 'val1','writetest_column:col2': 'val2'})


        while True:
            try:
                workers_table = connection.table('workers_table')
                workers_table.put(str(myPublicIp)+str(threadID),
{'workers_column:status': 'working'})
                break
            except Exception as e:
                print(e)

        print("Waiting for master...")

        listenFunction(connection)
```

```
        print("Next step... ThreadID: "+str(threadID))

        while True:
            try:
                status_table = connection.table('status_table')
                status_column = status_table.row('status_row')

stressTestType=status_column[b'status_column:type'].decode('ascii')
                break
            except Exception as e:
                print(e)

        timeStart = datetime.now()
        timeStampsList =[]

        while datetime.now() <
(timeStart+timedelta(seconds=int(liveTime))):

            if stressTestType == "read":

                timeStampsList.append(readFunction(connection))
            # elif stressTestType =="write":
            else :
                timeStampsList.append(writeFunction(connection,b))


        while True:
            try:
                timeStamps_table =
connection.table('timeStamps_table')
                timeStamps_table.put(str(myPublicIp)+str(threadID),
{'timeStamps_column:nQuery':
str(len(timeStampsList)),'timeStamps_column:avgQuery':str(mean(timeSta
mpsList))})
                workers_table.put(str(myPublicIp)+str(threadID),
{'workers_column:status': 'completed'})
                break
            except Exception as e:
                print(e)

        print(str(len(timeStampsList)))
        print(str(mean(timeStampsList)))
```

El comportamiento de las funciones mainFunction y mainFunction2 es similar al del resto de scripts.

Al igual que con MongoDB la librería del conector, en este caso *HappyBase*, tiene sus propias funciones y no ejecuta comandos como si se estuviera en la consola del servidor.

En cierto modo esto ha sido un problema o más bien un inconveniente, pues la sintaxis de Hbase de por sí no es muy densa y apenas tiene funcionalidades para hacer las tareas más básicas en una base de datos.

Aunque en este código esto no ha resultado ser un problema, pero sí, como se verá a continuación, en el código master.

Otro punto diferente es, a la hora de preparar las *queries* para las pruebas de escritura, en vez de preparar una variable que es una cadena de texto, se crea un objeto de tipo *batch* al que se le van insertando las instrucciones.

Otro punto diferencial es que cuando se realizan las *queries* de lectura, la información que se almacena es de tipo Byte por tanto para acceder al valor hay que decodificarlo y darle formato ASCII.

Finalmente, hacer notar que se han usado bucles de repetición en caso de salta excepción como en otros scripts anteriores y que se han creado los conectores dentro de las funciones. Realmente no se puede verificar la necesidad de haber hecho esto ya que se implementó así desde un principio.

```
def listenFunction(connection):

    status = "false"
    while status == "false":
        try:
            status_table = connection.table('status_table')
            status_column = status_table.row('status_row')

status=status_column[b'status_column:start'].decode('ascii')
        except Exception as e:
            print(e)

def readFunction(connection):

    timeStamp0=datetime.now()
    connection.table('readtest_table')
    return((datetime.now()-timeStamp0).total_seconds())

def writeFunction(connection,b):

    try:

        timeStamp0=datetime.now()
        b.send()
        return((datetime.now()-timeStamp0).total_seconds())

    except Exception as e:
        print("\n"+str(e)+"\n")
        pass
```

Estas funciones actúan exactamente igual que en los otros scripts.

La única diferencia en el caso de la función writeFunction es, que como se comentó anteriormente, la *query* preparada no es una cadena de texto sino un objeto *batch* que se envía a la base de datos y esta los procesa.

```
maxThread = os.cpu_count()*5

if maxThread < nThreads and nThreads > 0:

    print("You have exceed threads limit, it has been set to
"+str(maxThread)+".")
    nThreads = maxThread

if nThreads > 0:

    mainFunctionArgs = list(range(0,nThreads))
    with ThreadPoolExecutor() as executor:
        results = executor.map(mainFunction2, mainFunctionArgs)

else:
    mainFunction()
```

El final del código es exactamente igual para todas las bases de datos.

stmasterHBASE.py

```
import happybase
from datetime import datetime,timedelta
import time
import argparse
from statistics import mean

parser = argparse.ArgumentParser()
parser.add_argument("-ip", dest="ip", help="introduce the public IP of
the master instance")
parser.add_argument("-t", dest="type",default="read",help= "introduce
what type of stress test to perform: read or write")

# publicIpServer = parser.parse_args().ip
# stressTestType = parser.parse_args().type
stressTestType = "write"
publicIpServer = "3.142.133.138"
```

Esta parte es igual en todos los scripts.

```
def startFunction():

    print("Starting...")
    while True:
        try:
            connection = happybase.Connection(publicIpServer)
            break
        except Exception as e:
            print(e)

    status_table = connection.table('status_table')
    status_table.put('status_row', {'status_column:type':
str(stressTestType)})
    status_table.put('status_row', {'status_column:start': 'true'})
    workers_table = connection.table('workers_table')
    nWorkers = len(dict(workers_table.scan()))

    print(str(nWorkers)+" users will participate.")
    print("Stress Test type: " + str(stressTestType))
```

```
def measureFunction():

    while True:
        try:
            connection = happybase.Connection(publicIpServer)
            break
        except Exception as e:
            print(e)

    print("Measuring...")
    end = "false"
    while end == "false":

        workers_table = connection.table('workers_table')
        scan = dict(workers_table.scan())
        values_list=list(scan.values())
        uniques = list(set( val for dic in values_list for val in
dic.values()))
        if len(uniques) == 1 and uniques[0].decode('ascii') ==
"completed":
            end = "true"

    timeStamps_table = connection.table('timeStamps_table')
    scan = dict(timeStamps_table.scan())
    values_list=list(scan.values())
    values =[]
    for dic in values_list:

values.append(float(dic.get(b'timeStamps_column:avgQuery').decode('asc
ii')))

    avg = mean(values)
    print(avg)

    status_table = connection.table('status_table')
    status_table.put('status_row', {'status_column:start': 'false'})


    tables = ['workers_table','timeStamps_table','writetest_table']
    for table in tables:
        deleteTable(table,connection)
```

El comportamiento de amabas funciones es el mismo que en el resto de script.

El problema que surge (comentado en la explicación del código *worker*) es que la librería *Happybase*, para acceder a la información de las bases de datos tiene un sistema muy pobre de filtrado. Es obligatorio indicar la id de la fila a filtrar. Esto sumado a que no existe la agregación count() para contar número de filas o drop() para borrar todos los datos de una tabla, hace que la tarea de contar el número de usuarios o borrar información sea bastante tediosa.

Para leer información, lo que se hace con esta librería es, a través del conector escanear la tabla y guardar toda la información. Como no se puede

filtrar por columnas sin indicar una id específica hay que escanear toda la tabla y en el caso de la prueba de mil usuarios tiene que almacenar las mil filas y posteriormente calcular la longitud usando la función de Python len().

Por otro lado, no existe la forma de borrar toda la información de una tabla si no es borrando la tabla y creándola de nuevo. Solamente se puede borrar fila a fila por ello, se ha añadido una función adicional en este código para facilitar la limpieza de las tablas.

```
def deleteTable(table,connection):

    table = connection.table(table)
    scan = dict(table.scan())
    # print("FILAS A BORRAR:"+str(len(scan)))
    b=table.batch()
    row_key_list = list(scan.keys())
    for row_key in row_key_list:
        b.delete(row_key)
    b.send()
```

La función recibe como argumentos la tabla que se desea borrar y la conexión.

Primero con los argumentos recibidos se conecta a la tabla.

La escanea.

Prepara un objeto *batch* para realizar múltiples comandos a través de una sola petición.

Hace una lista con todas las *keys* de las filas de la tabla.

Entra en un bucle for que recorre cada fila de la tabla y añade al objeto *batch* el comando de borrar la fila.

Finalmente, se envía el objeto *batch* al servidor y borra todas las filas de la tabla.

```
startFunction()
measureFunction()
```

Se llaman a las funciones en el orden requerido.

3.3. Ejecución de las pruebas de estrés.

Ya hemos visto cómo se han configurado los equipos y cómo funcionan los scripts. En esta sección veremos con detalle los pasos a seguir para la ejecución de las pruebas de estrés.

La mejor forma de verlo es a través de un ejemplo, así que veremos distintas imágenes de cómo se ha realizado la prueba de estrés a la base de datos Neo4J.

Lo primero es poner la instancia de la base de datos y la instancia base en funcionamiento:

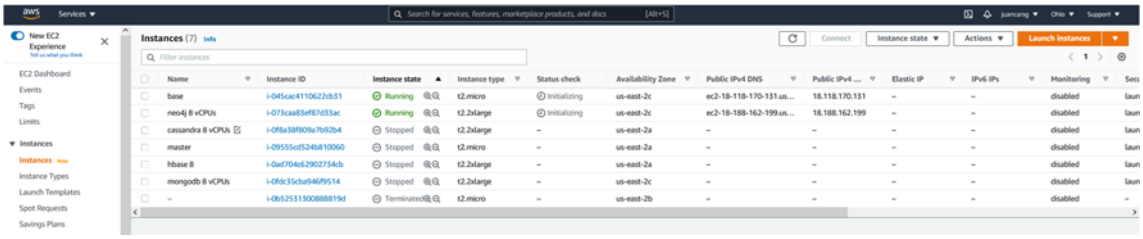

Imagen 27. Vista del panel de instancias de AWS EC2-Experience.

Lo siguiente es actualizar la instancia base con el script stworkerNEO4J para asegurarnos de que tiene la última versión de este. Para ello hay que abrir una consola y enviarle el archivo a través del protocolo SCP:

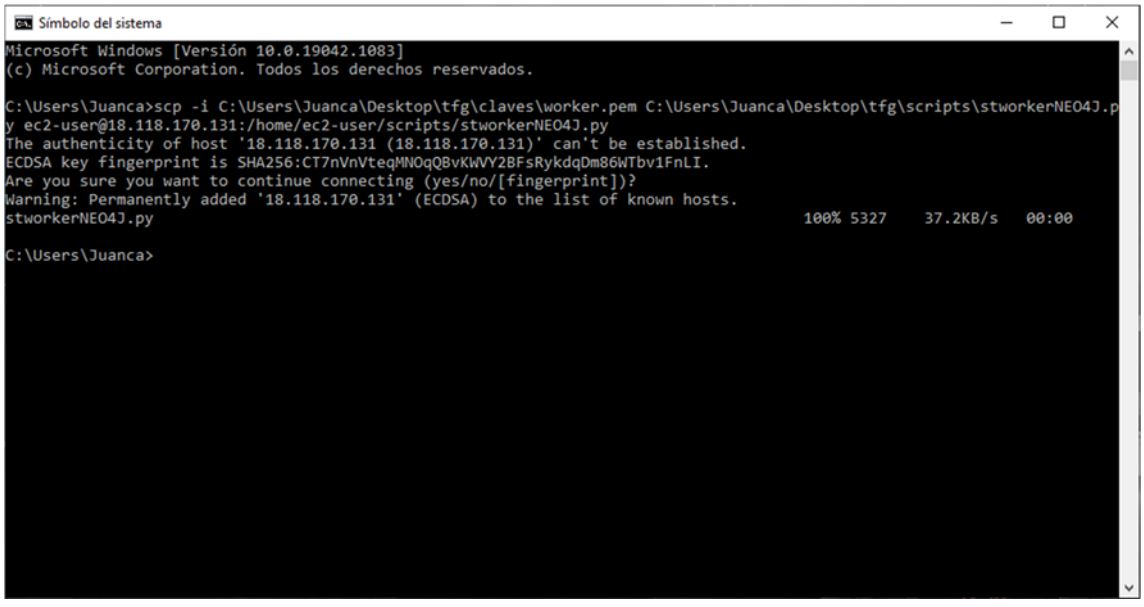

Imagen 28. Envió de script usando protocolo SCP.

En el caso de la instancia Neo4j, hay que iniciar el servidor de la base de datos ya que no está configurado para que se arranque con el inicio. Para ello hay que acceder remotamente a la instancia a través del protocolo SSH:

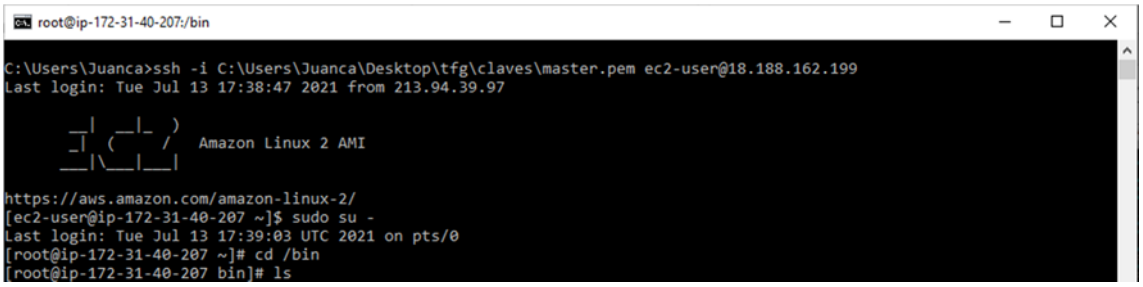

Imagen 29. Puesta en marcha de servidor Neo4j (I).

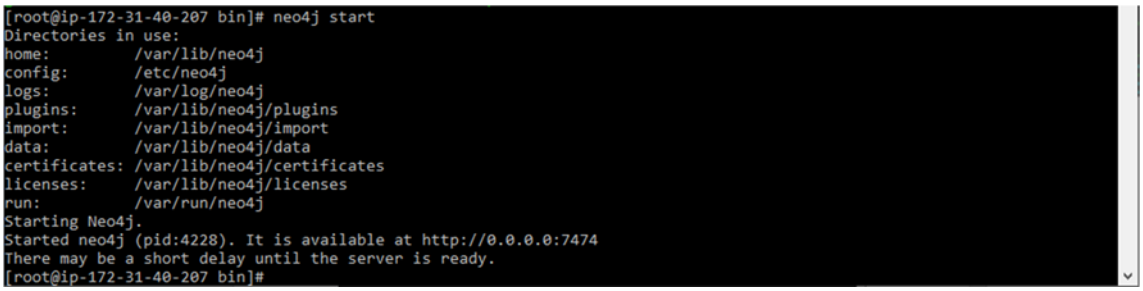

Imagen 30. Puesta en marcha de servidor Neo4j (II).

Para replicar las instancias worker a partir de la base primero hay que actualizar la AMI.

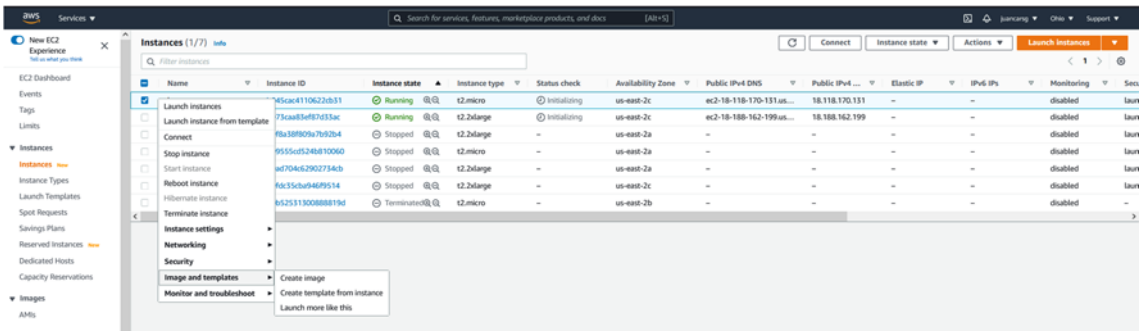

Imagen 31. Actualizar AMI base (I).

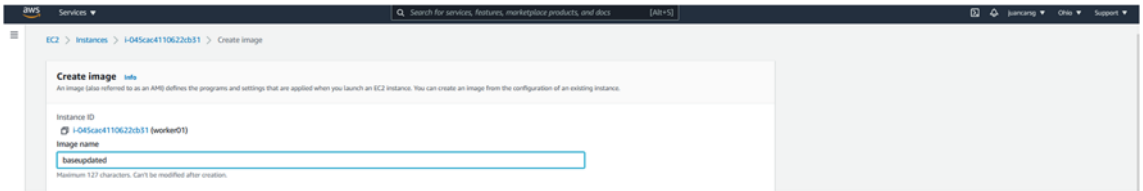

Imagen 32. Actualizar AMI base (II).

Con la AMI actualizada procedemos a crear las doscientas instancias. Hay que hacerlo en tres pasos. Primero se lanzará una única instancia con dos tags, Unique y Worker.

Después se lanzarán cien con dos tags, Hundred y Worker.

Finalmente se lanzarán las noventa y nueve restantes con un único tag, Worker.

Importante seleccionar el IAM EnablesEC2ToAccessSystemManagerRole para poder usar el servicio Run Command.

Una vez lanzadas hay que esperar a que se inicien todas.

Una buena forma de ver cuando están iniciadas es a través del dashboard:

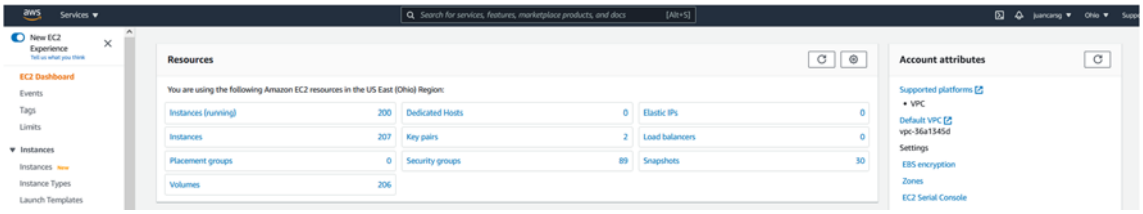

Imagen 33. Dashboard.

Cuando están todas en funcionamiento hay que ir a la aplicación Run Command.

El primer comando a ejecutar es el siguiente:

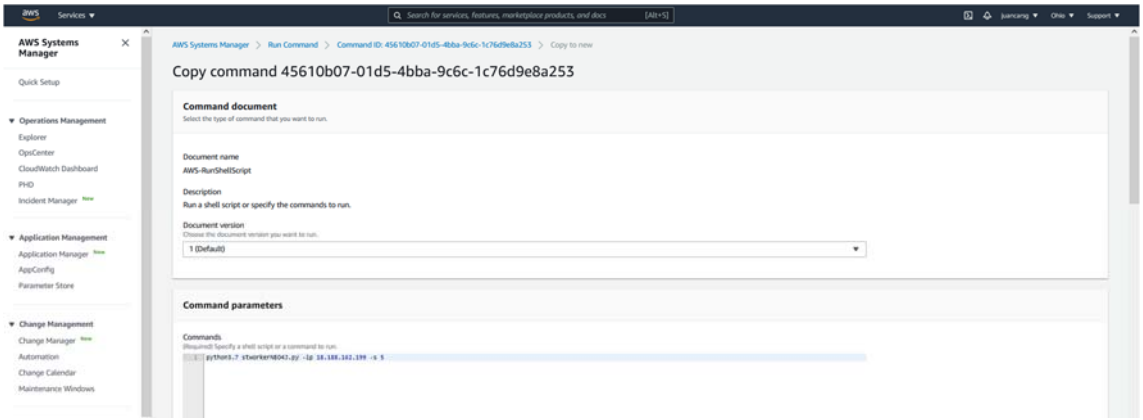

Imagen 34. Configuración de Run Command para prueba de estrés de un usuario (I)

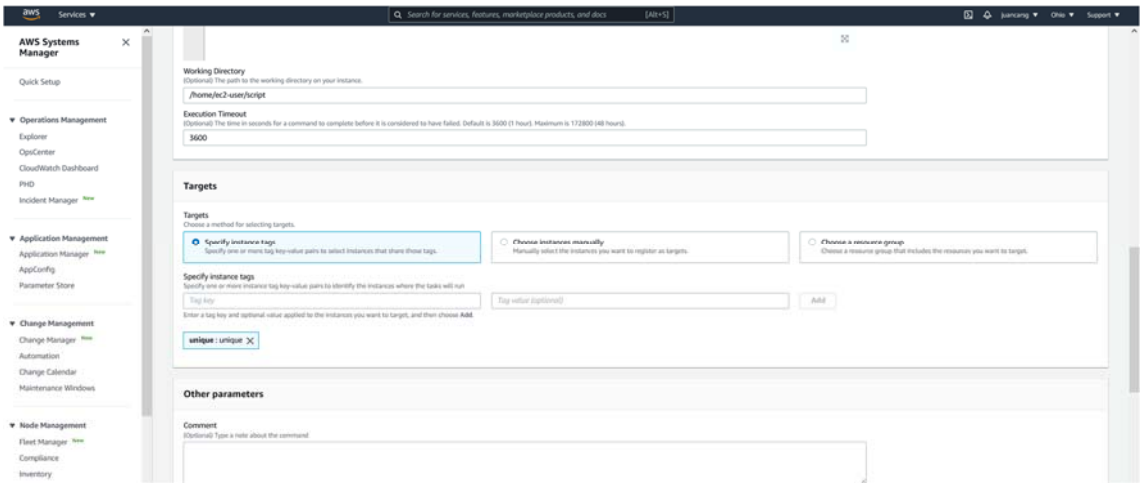

Imagen 35. Configuración de Run Command para prueba de estrés de un usuario (II)

Después ejecutaremos el script stmasterNEO4J.py y se efectuará la prueba.

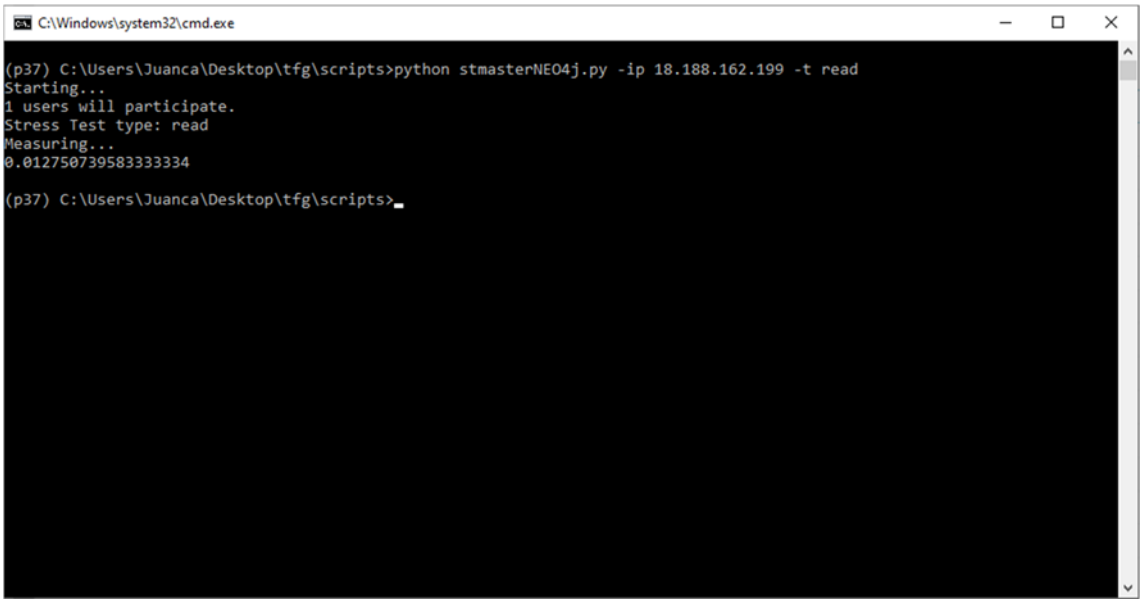

Imagen 36. Resultado prueba de lectura con un usuario.

Para repetir la prueba para el tipo escritura, en Run Command hay que repetir el comando. Puede hacerse rápidamente usando el botón Rerun.

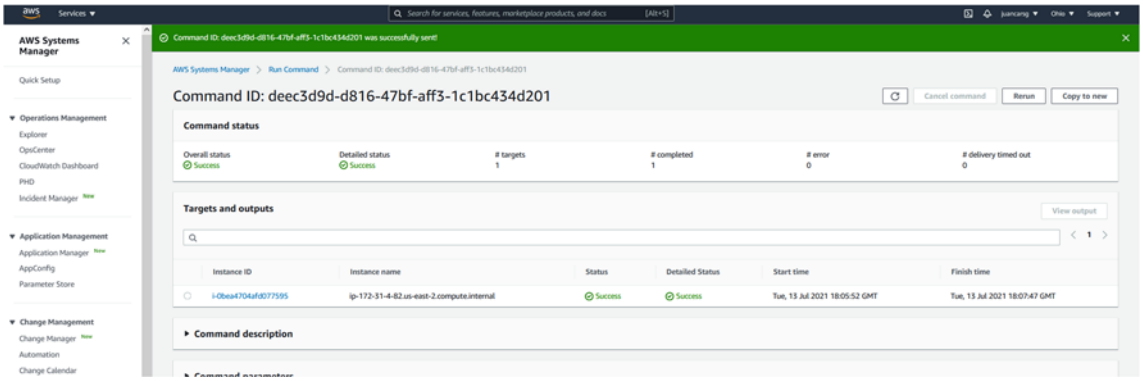

Imagen 37. Boton Rerun arriba a la izquierda para volver a ejecutar un comando.

Después volveremos a ejecutar el script master especificando la prueba de escritura.

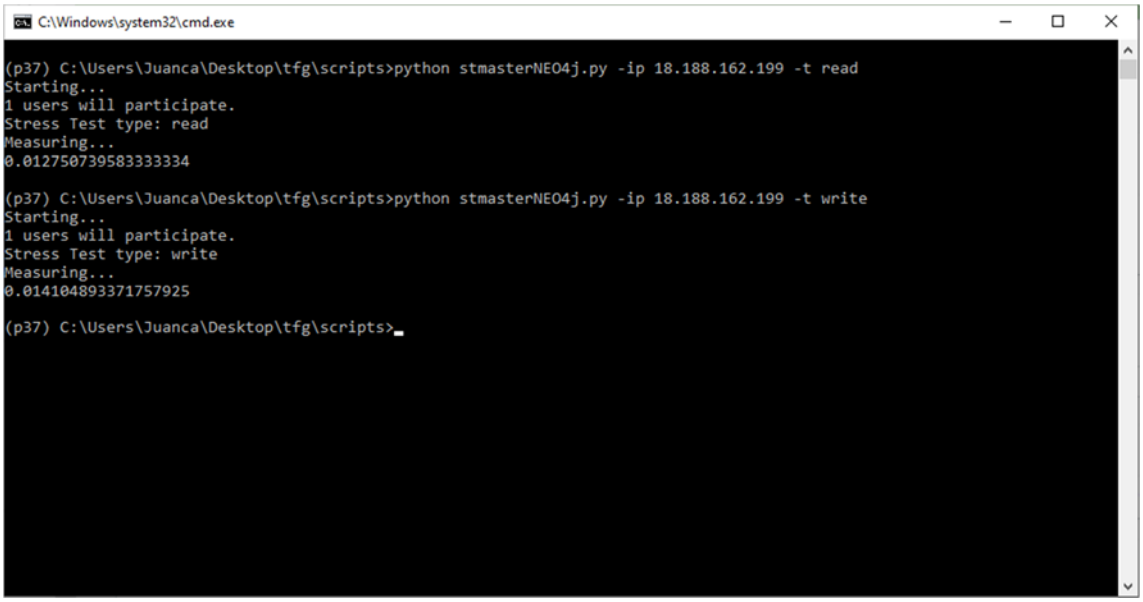

Imagen 38. Resultado prueba de escritura con un usuario.

Antes de volver a lanzar nuevamente el script *worker*, en Run Command hay que cambiar el target a las instancias con el tag hundred y establecer el porcentaje a cien en Rate Control:

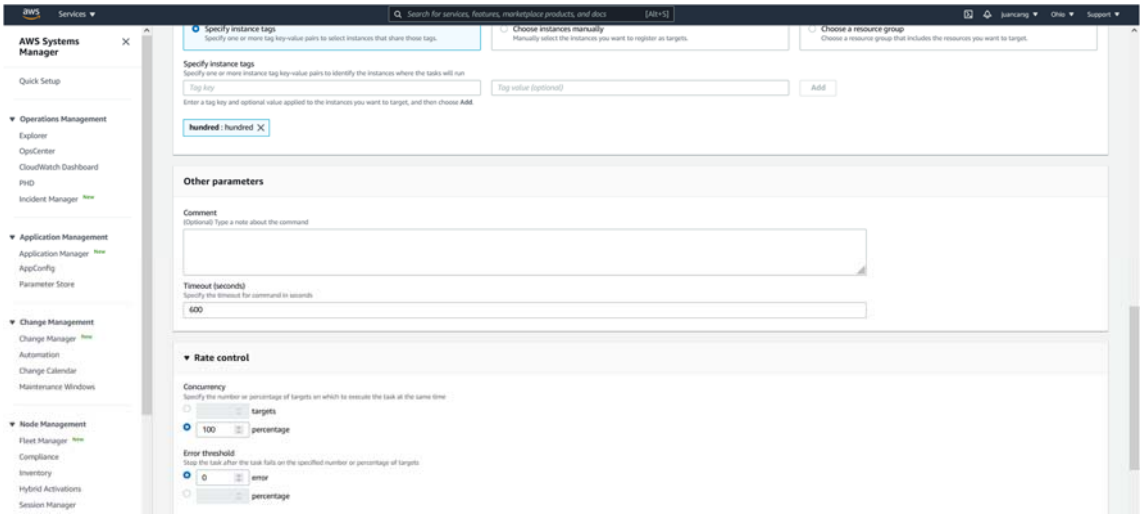

Imagen 39. Configuración de Run Command para prueba de estrés de cien usuarios.

Después volveremos a ejecutar el script master especificando prueba de lectura:

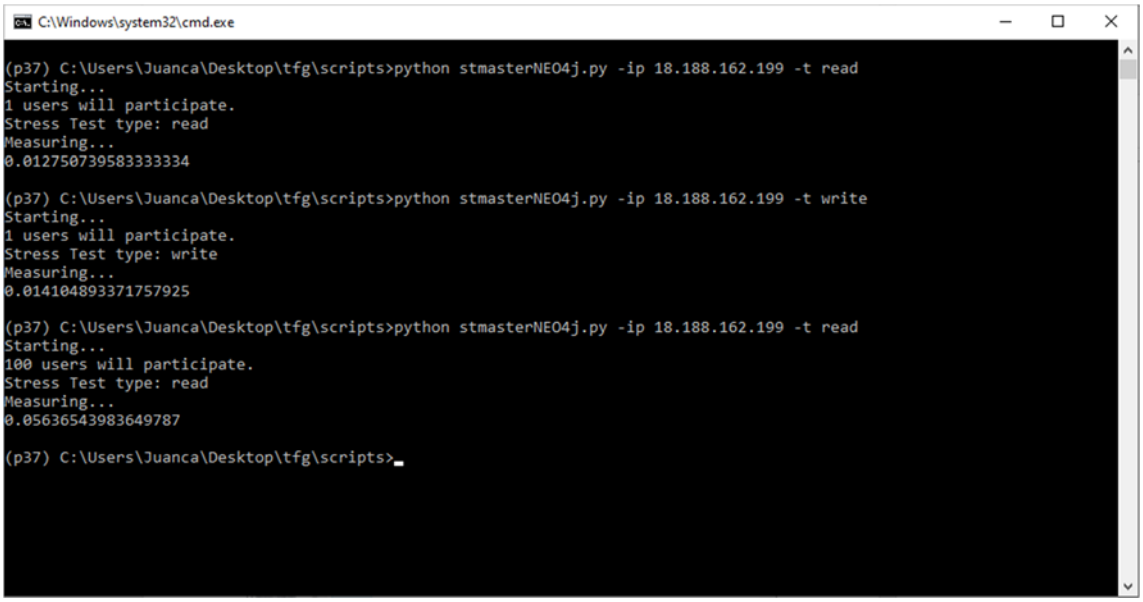

Imagen 40. Resultado prueba de lectura con cien usuarios.

En Run Command volveremos a usar *Rerun* y volveremos a lanzar el script master especificando prueba de escritura:

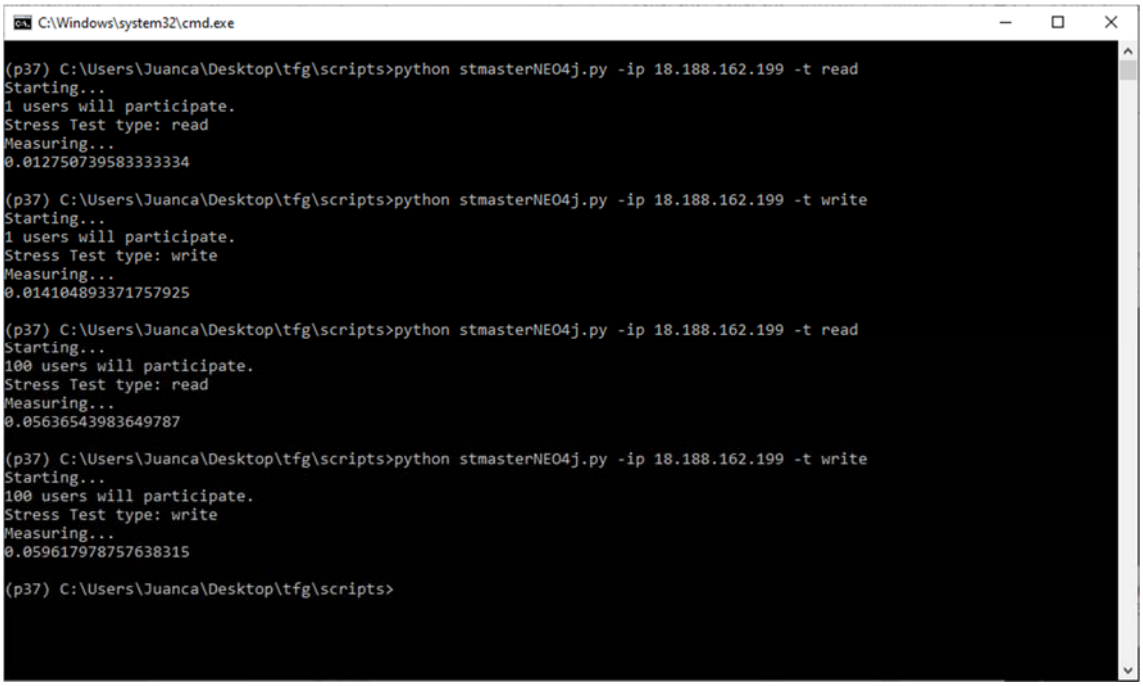

Imagen 41. Resultado prueba de escritura con cien usuarios.

Finalmente, para la prueba de mil usuarios, en Run Command hay que añadir el argumento -t 5 para especificar que se simulen cinco usuarios en cada instancia y cambiar las etiquetas objetivo y usar Worker.

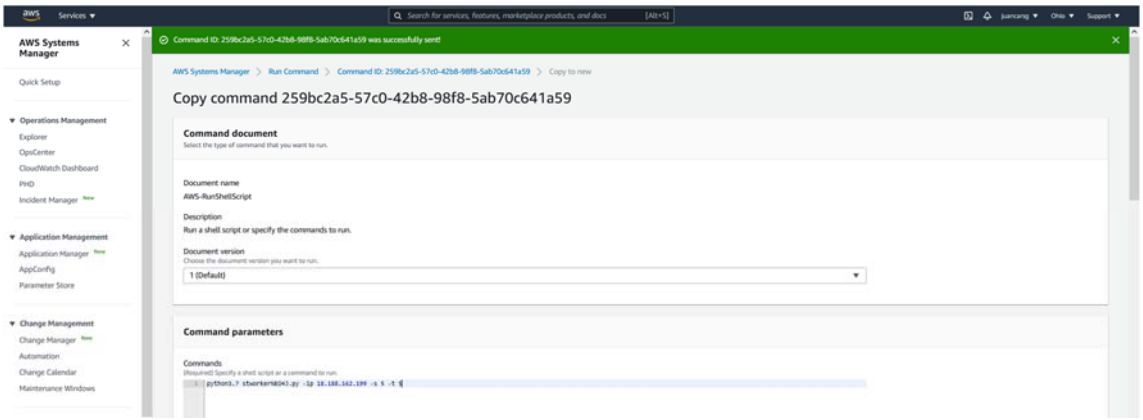

Imagen 42. Configuración de Run Command para prueba de estrés de mil usuarios (I).

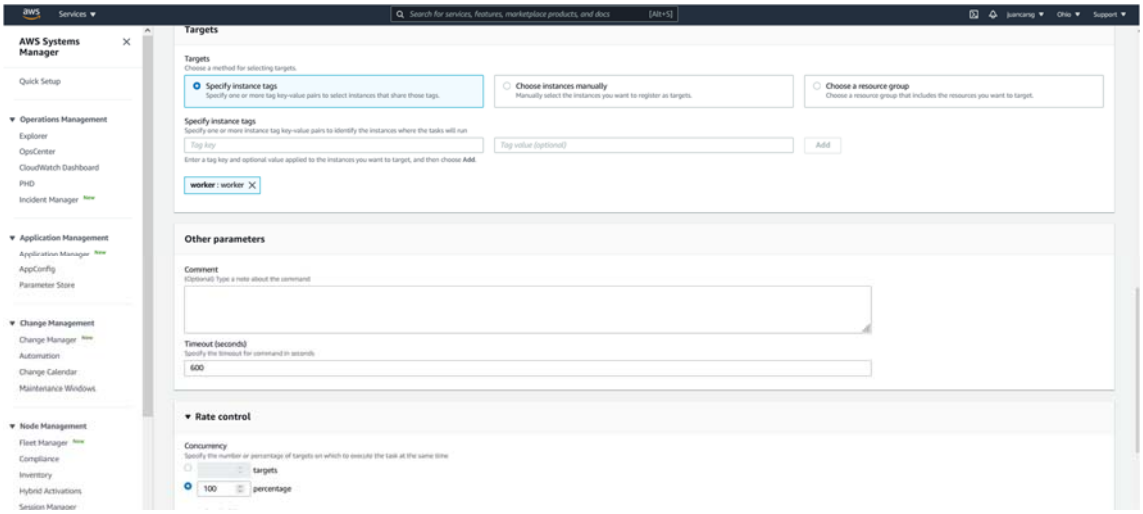

Imagen 43. Configuración de Run Command para prueba de estrés de mil usuarios (II).

Después hay que esperar unos segundos para que se registren todos los *workers* y ejecutar el script master especificando prueba de lectura:

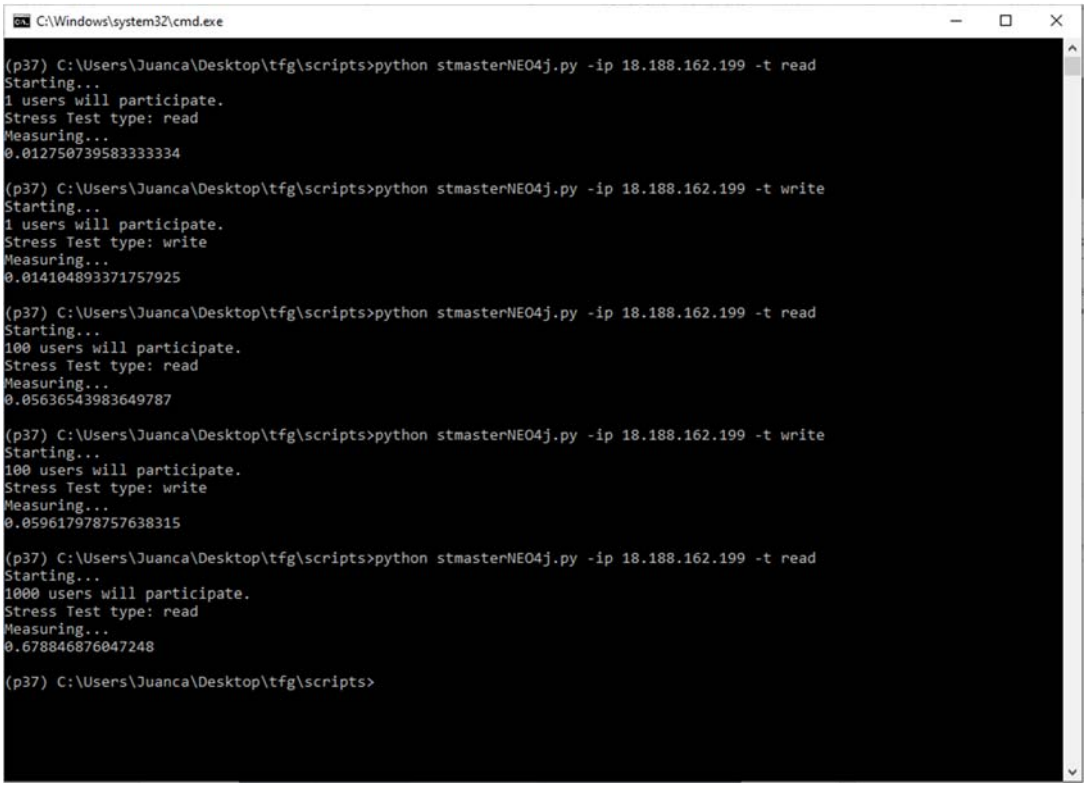

Imagen 44. Resultado prueba de lectura con mil usuarios.

Finalmente, hacer *click* en *Rerun* y ejecutar nuevamente el script maestro especificando prueba de escritura.

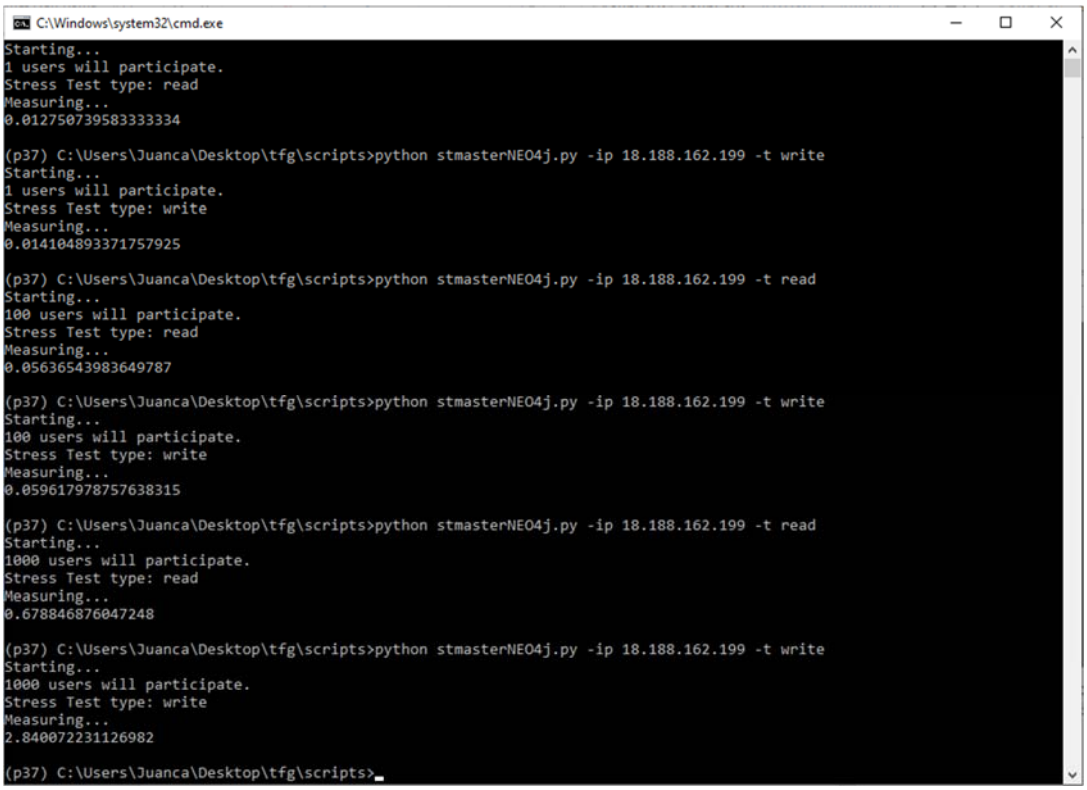

Imagen 45. Resultado prueba de lectura con mil usuarios.

Estos son todos los pasos que he seguido para llevar a cabo la ejecución de las pruebas. En el siguiente capítulo veremos los resultados.

Capítulo 4. Resultados de los Stress Test.

Los resultados obtenidos de todas las pruebas realizadas son los siguientes:

Tabla 2. Resultados obtenidos de las pruebas de estrés en segundos.

Tipo de prueba	MongoDB	Cassandra	Neo4J	HBase
Lectura 1 Usuario	0.004722334	0.006334836	0.012750739	0.000003398
Escritura 1 Usuario	0.006065389	0.004308768	0.014104893	0.000002939
Lectura 100 Usuarios	0.004622536	0.012023063	0.056365439	0.000003299
Escritura 100 Usuarios	0.004622536	0.010324390	0.059617978	0.000002635
Lectura 1000 Usuarios	0.023810525	0.123732894	0.678846876	0.000017816
Escritura 1000 Usuarios	0.025418194	0.0944113507	2.840072231	0.000014126

En la siguiente sección analizaremos estos datos y elaboraremos las conclusiones.

4.1. Opiniones personales.

Antes de analizar los resultados obtenidos y concluir en qué base de datos es la más efectiva para su uso en Big Data, me gustaría dar unas opiniones personales sobre cada una de ellas. Comentando, esencialmente, cómo ha sido mi experiencia durante la creación de los Scripts.

Si bien lo que realmente importa para calcular la efectividad de una base de datos de cara a ponerla en producción son las métricas obtenidas en la sección anterior, creo que también hay que tener en consideración las facilidades que se dan al desarrollador para crear aplicaciones que interactúen con la base de datos.

Quiero aclarar que las opiniones que voy a dar son desde el punto de vista de alguien que quiere usar la base de datos para crear aplicaciones usando el lenguaje Python. Desconozco de las herramientas disponibles para otros lenguajes de programación.

Cuando digo herramientas me refiero a librerías, y es que, en este aspecto, Python goza de una inmensa comunidad por lo que no resulta muy difícil encontrar librerías con funciones que te permitan realizar lo que busques.

MongoDB.

En el caso de MongoDB, la librería *pymongo* ofrece funciones que facilitan mucho la realización de consultas a la base de datos. Lo hemos visto durante la explicación de los códigos. Con MongoDB no se ha requerido tratar de ninguna forma la información obtenida para llegar al dato deseado una vez realizada la *query*. Esto se debe en gran parte a lo rica que es la sintaxis de la base de datos. En definitiva, esto es un punto muy grande a su favor.

Por otro lado, la existencia de la herramienta Mongo Compass facilita también la tarea de visualizar la información. Aunque existen programas que hacen lo mismo con cualquier base de datos, se trata de programas de terceros por lo que hay que comprar licencias y esto no es nada agradable.

También, la flexibilidad que aporta MongoDB la hora de insertar la información sin necesidad de haber preestablecido las columnas ni los tipos de valores es algo muy positivo a destacar.

Por último, algo negativo, que realmente no lo es del todo, pero sí que puede serlo al principio cuando no conoces la sintaxis de la base de datos. Estoy hablando de la sintaxis en sí. De las cuatro bases de datos que he estudiado, MongoDB es con diferencia la que mas se aleja del lenguaje SQL con el cual estoy más familiarizado.

Cassandra.

Después de estudiar el modelo clave-valor con Cassandra y el modelo familia de columnas con Hbase, modelos que, por cierto, son muy similares, me he dado cuenta que están diseñados para almacenar la información de una forma distinta a la que estoy acostumbrado. Me he podido dar cuenta de esto al ver que en la sintaxis que no tienen agregaciones básicas para analizar la información de manera vertical. Hablo por ejemplo de la función distinc(). Este punto es algo que ha resultado ser una molestia durante el desarrollo de los scripts.

Aun así, en el caso de Cassandra, me ha gustado que la librería permita la ejecución de comandos como si estuviera en la consola del servidor ya que, de esta forma, he tenido que aprender sólamente la sintaxis de Cassandra y no la sintaxis y cómo usar las funciones de la librería.

Neo4j.

Neo4j es sin lugar a dudas la base de datos que más se aleja del modelo relacional.

Me ha parecido muy interesante la forma de clasificar la información mediante nodos y etiquetas.

Creo que su sintaxis es muy rica e intuitiva y de las cuatro que he estudiado es la que me ha resultado más fácil entender y aprender.

Esto último junto con el hecho de que la librería al igual que con Cassandra ejecute las *queries* como si se estuviera en la consola del servidor es algo que se aprecia mucho.

Hbase.

Hbase ha demostrado en los resultados tener un gran potencial. El problema que le veo, al menos desarrollando en Python, es que la librería *HappyBase* no incorpora funciones para algunas agregaciones del lenguaje de Hbase.

Esto agrava más el hecho de que Hbase de por sí tiene un lenguaje muy pobre.

4.2. Métricas.

Salta a la vista que los mejores resultados los ha obtenido Hbase. Pero antes de llegar a la conclusión final, vamos a analizar la tabla de resultados.

Como se puede ver, los resultados obtenidos no varían mucho en los escenarios de un usuario y cien usuarios con MongoDB y Hbase.

Por lo que la primera conclusión que se puede sacar es que la carga de cien usuarios no parece afectarles a estas dos bases de datos.

Al contrario que con Cassandra y Neo4j que si afecta negativamente en los resultados.

Por otro lado, algo interesante que se puede apreciar es que, de forma general, las pruebas de escritura y lectura requieren un tiempo similar de procesamiento. A excepción de Neo4j, que cuando hay mil usuarios escribiendo se nota una subida muy notable del tiempo requerido en la prueba de escritura.

Por lo que otra conclusión sería que el modelo gráfico no rinde bien al haber muchos usuarios insertando nodos.

Por último, y más evidente, Hbase ha resultado ser la que mejor rinde de todas con mucha diferencia.

Creo que esto se debe en gran parte a cómo esta diseñado el modelo. Que a costa de perder interactividad a la hora de manipular las consultas para obtener un dato especifico, se ha logrado incrementar notablemente la efectividad de procesado de las peticiones.

Para aplicaciones Big Data, esta base de datos es evidentemente la mejor, ya que se requerirán insertar varios cientos de miles incluso millones de datos y en este contexto de escritura, sería super eficaz.

El problema que podría surgir, en el contexto de lectura, es que, como mencioné en las opiniones personales, el lenguaje es muy pobre y analizar los datos de manera correcta requerirá del uso de otras herramientas.

Pero, en definitiva, Hbase es la mejor base de datos de las estudiadas.

Capítulo 5. Conclusiones.

En conclusión, durante el desarrollo del proyecto he podido poner en práctica algunos conocimientos adquiridos a lo largo de mis estudios además de aprender nuevos.

Empezando por descubrir el servicio AWS que sin lugar a dudas me ha abierto un sinfín de puertas de cara a empezar nuevos proyectos en un futuro.

He aprendido a crear y configurar máquinas con distribuciones Linux lo cual, siempre me ha supuesto un reto durante mi etapa como estudiante y creo que a raíz de este proyecto he conseguido sintetizar mis conocimientos y sentirme más cómodo con el sistema operativo.

También he aprendido muchas funcionalidades del lenguaje de programación Python, que hasta el momento no había tocado.

En definitiva, pienso que realizar este proyecto ha resultado ser una experiencia muy enriquecedora porque, por un lado, el hecho de haber programado los scripts, el haber tenido que modificarlos para adaptarlos a cada base de datos y el tener que buscar soluciones a cada problema que me fue surgiendo, ha potenciado mi capacidad de análisis y solución de errores. La cual creo que es muy importante para un desarrollador y la única forma de entrenar es a base de ensayo y error trabajando en cualquier proyecto.

Por otro lado, pienso que me ha abierto la mente y me ha aclarado mucho las ideas sobre cómo quiero continuar con mis estudios de posgrado y de alguna forma encaminar mi carrera.

Bibliografía y referencias.

1. La base de datos líder del mercado para aplicaciones modernas | MongoDB. Accessed July 14, 2021. https://www.mongodb.com/es
2. Documentation. Accessed July 14, 2021. https://cassandra.apache.org/doc/latest/
3. Graph Database Platform | Graph Database Management System | Neo4j. Accessed July 14, 2021. https://neo4j.com/
4. Apache HBase – Apache HBase™ Home. Accessed July 14, 2021. https://hbase.apache.org/
5. Tipos de instancias de Amazon EC2 - Amazon Web Services. Accessed July 11, 2021. https://aws.amazon.com/es/ec2/instance-types/
6. Quick Setup Host Management - AWS Systems Manager. Accessed July 12, 2021. https://docs.aws.amazon.com/systems-manager/latest/userguide/quick-setup-host-management.html
7. Step 4: Create an IAM instance profile for Systems Manager - AWS Systems Manager. Accessed July 12, 2021. https://docs.aws.amazon.com/systems-manager/latest/userguide/setup-instance-profile.html
8. Install MongoDB Community Edition on Amazon Linux — MongoDB Manual. Accessed July 12, 2021. https://docs.mongodb.com/manual/tutorial/install-mongodb-on-amazon/
9. Deploy Neo4j using the Neo4j RPM package - Operations Manual. Accessed July 12, 2021. https://neo4j.com/docs/operations-manual/current/installation/linux/rpm/#linux-rpm-install
10. Apache HBase ™ Reference Guide. Accessed July 13, 2021. https://hbase.apache.org/book.html#getting_started
11. concurrent.futures — Launching parallel tasks — Python 3.9.6 documentation. Accessed July 13, 2021. https://docs.python.org/3/library/concurrent.futures.html#