UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Análisis de rendimiento de Migración de Datos a MongoDB
Autores: Marcelo Juan Herrón
Director: Fernando G. Tinetti
Codirector: -
Asesor profesional: Franco Agustín Terruzzi
Carrera: Licenciatura en Sistemas

Resumen

Las bases de datos relacionales son sistemas conocidos y robustos que han sido usados desde su primera aparición en los años 70 para el almacenamiento y recuperación de datos. No obstante, en los últimos años han comenzado a ganar popularidad las bases de datos NoSQL, nacidas por las necesidades de la web moderna de bases de datos más flexibles.
En este contexto, este trabajo plantea el análisis a una base de datos NoSQL en particular: MongoDB, una de las más reconocidas y utilizadas actualmente. El interés principal de este trabajo se centra en su rendimiento para almacenar distintas cantidades de datos. Este análisis se llevará a cabo por medio de una serie de experimentos dedicados a obtener un muestreo de tiempos para poder así establecer conclusiones.
Con ese objetivo se propone utilizar MongoDB combinado con las herramientas Mongoose y Express, los cuales en conjunto forman una de las posibles arquitecturas de software "completas" alrededor de MongoDB.
Se espera así identificar relaciones entre cada elemento que conforma la plataforma propuesta en cuanto a su impacto general en cuanto a rendimiento de las operaciones sobre la base de datos y discriminar dónde se encuentran los principales cuellos de botella si los hubiera.

Palabras Claves

Bases de datos – NoSQL – Análisis de rendimiento - Mongoose - Express – API REST - Capas de abstracción – Pruebas de rendimiento

Trabajos Realizados

Desarrollo de scripts pensados para medir los tiempos de inserción de datos de cada subsistema de la plataforma, teniendo en cuenta y documentando la configuración necesaria para obtener los mejores resultados. Además fueron necesarias soluciones adicionales para el chequeo de consistencia de los datos manipulados y medidas para prevenir posibles fallos.
Con la definición de estas nuevas herramientas se realizaron pruebas de rendimiento para cada elemento de la arquitectura. Dichas pruebas consistieron en la ejecución de los scripts anteriormente mencionados para distintas cantidades y tamaños de datos, analizando, describiendo y comparando los resultados obtenidos en cada caso.

Conclusiones

Al abstraer cada componente de la plataforma como una capa que agrega funcionalidad al sistema de software completo, se pudo analizar el rol e impacto que tiene cada uno de los mismos en cuanto a los tiempos agregados de cada operación y se obtuvo además una idea de dónde se encuentran los principales cuellos de botella. Luego de haberse implementado las soluciones para realizar las pruebas necesarias sobre cada uno de estos componentes y ejecutado dichas pruebas, se pudieron obtener resultados relevantes frente a los objetivos propuestos y distintos a los inicialmente estimados.

Trabajos Futuros

Estudio comparativo entre MongoDB y una base de datos relacional; Análisis del problema de configuración de MongoDB; Mejora sobre los algoritmos utilizados

Fecha de la presentación: Octubre, 2018

Dedicatoria

A mis padres Alejandro y Claudia por creer siempre en mí y haberme acompañado en cada etapa de mi vida.
A mis hermanos por el apoyo incondicional y por ayudarme y motivarme a ser una mejor versión de mí mismo.
A mis amigos por estar siempre conmigo, en las buenas y en las malas.

Agradecimientos

A la Facultad de Informática de la Universidad Nacional de La Plata, no solo por permitir mi formación académica sino también por brindar un excelente ámbito intelectual y al mismo tiempo cálido y familiar.
A mi director de tesis, el Dr. Fernando G. Tinetti, por su increíble paciencia y tiempo dedicado a la realización de esta tesis.
A mi asesor profesional, el Lic. Franco Agustín Terruzzi, por su amistad, paciencia y predisposición para todo tipo de ayuda que alguna vez necesité.
A mis amigos y compañeros con quienes compartí tanto tiempo juntos, incluyendo aquellas largas noches de estudio sin sueño.
A toda mi familia por su apoyo incondicional durante mi carrera y por siempre haber puesto su fe en mí.

“Aquellos que pueden imaginar cualquier cosa, pueden crear lo imposible”

— Alan Turing (1912 - 1954)

Índice de Contenidos:

Capítulo 1 | Introducción, Motivación y Objetivos

Las bases de datos relacionales son sistemas conocidos y robustos que han sido usados desde su primera aparición en los años 70 [Quick Base, Inc, 2018]. De hecho, hasta la actualidad continúan siendo la forma más popular y utilizada para el almacenamiento y recuperación de datos. No obstante, en los últimos años han comenzado a ganar popularidad las bases de datos NoSQL, nacidas por las necesidades de la web moderna de bases de datos más flexibles.
En este contexto, este trabajo plantea el análisis a una base de datos NoSQL en particular: MongoDB, una de las más reconocidas actualmente y utilizada en producción desde hace algunos años. El interés principal de este trabajo es en particular su rendimiento para almacenar distintas cantidades de datos. Este análisis se llevará a cabo por medio de una serie de experimentos dedicados a obtener un muestreo de tiempos para poder así establecer conclusiones.
A lo largo de este capítulo serán planteados los lineamientos principales en la forma de objetivos y motivaciones, junto con una descripción general de los contenidos de este trabajo.

1.1. Objetivo

El objetivo de esta tesina es el de analizar el escenario, comportamiento, estrategias y rendimiento de la migración de grandes cantidades de datos hacia el entorno de una base de datos NoSQL utilizado MongoDB [MongoDB Inc, 2017a] como sostén de la infraestructura de software. Se propone utilizar MongoDB combinado con Mongoose [Mongoosejscom, 2018] (modelado y validación de datos) y Express [StrongLoop, IBM, 2018] (framework de aplicaciones web para node.js [Joyent, Inc, 2018]), los cuales en conjunto forman una de las posibles arquitecturas de software “completas” alrededor de MongoDB. Un aspecto importante a identificar lo más claramente posible es la relación entre las distintas capas de la plataforma en cuanto a su impacto general en lo que respecta al rendimiento de las operaciones sobre la base de datos y discriminar dónde se encuentran los principales cuellos de botella si los hubiera.
Con lo mencionado en el párrafo anterior, se espera obtener resultados descriptivos del rendimiento de la base de datos MongoDB, permitiendo así actuar como una base para futuros estudios y establecer conclusiones sobre dichos resultados que puedan contribuir a la toma de decisiones en cuanto a qué sistema utilizar para casos particulares.
Además, mediante todo el proceso de desarrollo del testeo necesario se espera obtener conocimiento sobre la arquitectura así también como de los posibles problemas y errores comunes que pueden ocurrir, en especial aquellos ligados a la cantidad o tamaño de los datos.

1.2. Motivación

Dentro de las bases de datos NoSQL se eligió MongoDB en particular por ser una de las más conocidas y populares, usada de manera total o parcial por aplicaciones con centenas de miles o millones de usuarios como Twitter, Facebook, Google, Ebay y Baidu [MongoDB Inc, 2017c] entre otros. También cuenta con una comunidad activa, brindando soporte y una buena cantidad de documentación.

A pesar de todo esto, al ser un campo en actual desarrollo, resulta interesante la realización de pruebas y estudios que permitan aportar evidencia para respaldar algunas de las afirmaciones hechas por estos sistemas, en particular sobre su rendimiento para grandes cantidades de datos. Es ese interés el que motiva a hacer un estudio detallado, haciendo énfasis en los tiempos de cada capa de la plataforma y la sobrecarga generada por cada una de ellas para obtener así evidencia empírica que actúe como base para futuros proyectos.

1.3. Organización del Contenido

Con el fin de cumplir con los objetivos propuestos anteriormente, a lo largo de esta tesina se estableció un marco teórico que abarca todos los conceptos de la plataforma y tecnologías a utilizar. Entre los conceptos más importantes se encuentran: NoSQL, MongoDB, Mongoose.js, Express y aplicaciones REST. Luego fue necesario establecer las estructuras de datos que serían utilizadas para la ejecución de las pruebas.

Una vez establecidos todos estos conceptos como base, se procedió a implementar una serie de scripts basados en mongo shell [MongoDB Inc, 2017d] y node.js para cada una de las capas de la arquitectura en cuestión (MongoDB, Mongoose y Express). Dichos scripts fueron pensados para medir los tiempos de cada capa para los distintos tamaños de datos, teniendo en cuenta y documentando la configuración necesaria para obtener los mejores resultados. Además fueron necesarias soluciones adicionales para el chequeo de consistencia de los datos manipulados y medidas para prevenir posibles fallos.

Con la definición de estas nuevas herramientas se realizaron pruebas de rendimiento para cada capa de la arquitectura. Dichas pruebas consistieron en la ejecución de los scripts anteriormente mencionados para distintas cantidades y tamaños de datos, analizando, describiendo y comparando los resultados obtenidos en cada caso.

Se hizo énfasis además en los problemas ocurridos al momento de ejecutar dichas pruebas, en particular sobre aquellos propios de la tecnología utilizada que fueron encontrados y las configuraciones de la aplicación que fueron necesarias para solucionarlos o evitarlos. Finalmente se establecieron conclusiones sobre los resultados de dichas pruebas y sobre la plataforma en general, dando lugar así a la propuesta de varios posibles trabajos futuros a realizar a partir de las mismas.

En el transcurso de este trabajo serán abordados en detalle cada uno de los pasos de este proceso. A continuación se lista el resto de los capítulos que integran este documento, junto con un breve resumen del contenido de los mismos.

Capítulo 2:

El objetivo de este capítulo es introducir los conceptos básicos necesarios para establecer una base sobre los temas a tratar a lo largo del trabajo. Se hace énfasis en el concepto de base de datos NoSQL y sus diferencias principales con las bases de datos relacionales. Como parte de los conceptos que se utilizan en la experimentación serán definidos MongoDB y las ideas básicas de una interfaz REST.

Capítulo 3:

Una vez establecidos los conceptos básicos de las tecnologías utilizadas, en este capítulo se definen las estructuras de datos a utilizar en las pruebas y se detalla en profundidad la estructura que conforma la plataforma de trabajo. Cada una de las partes que conforman esta estructura forma el sistema en capas mencionado en el capítulo 1.

Capítulo 4:
Antes de realizar las pruebas de rendimiento sobre cada capa de la plataforma fue necesario establecer el método de inserción disponible más eficiente. En este capítulo se presentan benchmarks de dichos métodos y se justifica la elección de uno de ellos sobre el resto. Estas pruebas previas a las de rendimiento propiamente dichas sirvieron a su vez para encontrar la configuración “ideal” o la más eficiente sobre las posibles combinaciones de parámetros (tamaños de buffer y datos, cantidad de elementos, etc) para cada uno de los métodos de inserción. Por otro lado, este capítulo detalla uno de los problemas principales encontrados al momento de realizar la ejecución de inserciones de grandes cantidades de datos.

Capítulo 5:
En este capítulo se presentan los resultados obtenidos de las pruebas para cada una de las capas de la plataforma. Los resultados de cada capa son analizados individualmente en un principio y luego son comparados entre sí. Sobre los resultados se establecen además gráficos que ayudan a mostrar la tendencia de los tiempos de ejecución obtenidos. Finalmente en este capítulo se analizan las posibles razones por las cuales la ejecución de ciertas pruebas obtienen resultados distintos al de otras.

Capítulo 6:
En este capítulo se analiza en detalle el aspecto técnico de cada uno de los scripts desarrollados y utilizados para la realización de las pruebas de rendimiento. Para complementar dicho análisis se presentan versiones en pseudocódigo de los mismos, especificando detalles del funcionamiento, las consideraciones necesarias y ejemplos de ejecución para cada uno de ellos. El contenido de este capítulo puede parecer en desorden con respecto al capítulo anterior ya que la implementación de los algoritmos precede a la realización de las pruebas en sí. Sin embargo, la decisión de presentarlo de esta manera está dada para priorizar las pruebas de rendimiento, el foco principal de este trabajo.

Capítulo 7:
Como cierre de este trabajo, este capítulo presenta las conclusiones finales obtenidas a partir del desarrollo realizado y las obtenidas a partir de los resultados de las pruebas de rendimiento. Finalmente se establecen posibles líneas para trabajos futuros a ser realizados tomando a este como su base.

Capítulo 2 | Conceptos Básicos: noSQL, MongoDB y API REST

A lo largo de este capítulo se definirán los conceptos básicos sobre los cuales se trabajará a lo largo de esta tesina. Será introducido y definido el concepto de base de datos NoSQL, especificando características y tipos. A su vez, se analizará su diferencia con respecto a bases de datos relacionales, contemplando posibles ventajas y desventajas para distintos casos de uso. Dado que nos interesa el análisis con al menos un ejemplo implementado real se definirá MongoDB, la principal herramienta a utilizar y el foco de estudio de este trabajo.
Finalmente se definirá el concepto de aplicación REST, el cual determinará la estructura de la plataforma de trabajo. La decisión de modelar la aplicación de esta forma está dada en que la convierte en una interfaz con una entrada estándar, facilitando la reutilización de la plataforma para posibles usos futuros.
Se considera una base de datos noSQL a aquella que no sigue los principios de un RDBMS (Relational Data Base Management System) [IBM, 2010] y por lo general se relaciona a grandes conjuntos de datos accedidos y manipulados en escala Web. La razón detrás de esta relación, sin embargo, no está fundada en estudios empíricos y por lo general es una asociación producto de la naturaleza no estructurada de noSQL. Las características particulares de este tipo de bases de datos serán abordadas a lo largo de este capítulo.
Contrario a su nombre las bases de datos noSQL pueden hacer uso del lenguaje de consultas dependiendo el caso. Hoy en día el término es popularmente interpretado como “no solo SQL” y la principal implicancia de su nombre es su naturaleza no relacional.

2.1. Bases de datos relacionales y sus limitaciones

Desde su introducción en 1970 por Edgar Codd [Van Canneyt, 2005], las bases de datos relacionales han sido la forma de almacenamiento predominante en aplicaciones y organizaciones que requieren de un amplio almacenamiento de datos.
Entre sus principales ventajas se encuentran:

- Manejo de concurrencia: se realiza mediante transacciones lo cual ayuda a disminuir la complejidad sobre la concurrencia de la aplicación y permite volver a un estado anterior válido en caso de error durante una transacción.
- Modelo estandarizado: El lenguaje de consulta (SQL) suele ser muy similar entre distintos RDBMS, esto permite que pueda ser reutilizado constantemente sin la necesidad de volver a aprender un lenguaje nuevo desde el inicio.
- Facilidad de integración: Debido al modelo estandarizado, es posible compartir una base de datos entre varias aplicaciones con relativa facilidad. El manejo de la concurrencia en este caso será controlado de la misma manera que en el caso del acceso de múltiples usuarios de una única aplicación.

Los RDBMS tradicionales se focalizan en las propiedades ACID [IBM, 2018a], las cuales representan:

- Atomicidad: Todo cambio a los datos se realiza como una sola operación, es decir, o se realizan todos o no se realiza ninguno.
- Consistencia: La base no puede terminar en un estado inconsistente luego de una transacción (exitosa o no).
- Aislamiento (Isolation): Una transacción no puede interferir con otra, es decir que los estados intermedios de una transacción son invisibles para las demás.

- Durabilidad: Luego de una transacción exitosa el cambio persiste incluso luego de un reinicio o fallo del sistema.

2.1.1. Problemas del modelo relacional

Uno de los posibles problemas con las bases de datos relacionales es lo que se conoce como “impedance mismatch” o la diferencia de impedancias [Sadalage & Fowler, 2013a]. El término hace alusión a la diferencia que existe entre las estructuras de datos en memoria y el modelo relacional.
En el modelo relacional todo se encuentra definido como una tupla (un par nombre-valor) o como una relación (conjunto de tuplas) con la limitación de que los valores en una tupla son de tipo simple, es decir no pueden contener estructuras. Dicha limitación no existe en las estructuras de datos en memoria, por consiguiente es necesaria una traducción para su almacenamiento en la base de datos.
Eventualmente este problema fue aligerado por frameworks de mapeo de objetos a bases de datos relacionales como Hibernate [Red Hat Inc., 2018], sin embargo esta no es una solución definitiva al problema.
Por otro lado, con la gran y rápida expansión de la World Wide Web a partir de los años 90 [Berners-lee, 1996], que aún continúa en el presente, creció también el número de usuarios y el volumen de la información almacenada por las organizaciones.
Las bases de datos relacionales asumen que las propiedades de la información pueden ser definidas de antemano y que sus relaciones se encuentran claramente establecidas. Sin embargo la información irregular y no estructurada que se suele recolectar de los sitios web y sus usuarios complejiza su almacenamiento en un RDBMS.
Uno de los retos a escala web es, sin embargo, mantener tiempos de respuesta rápidos para el acceso y almacén de los datos sin sacrificar las propiedades transaccionales.

2.2. Orígenes de NoSQL

Con el objetivo de construir una infraestructura escalable para el procesamiento paralelo de grandes cantidades de datos y resolver así los problemas presentados por los RDBMS frente a estos casos, Google desarrolló un mecanismo en capas para atacar cada problema. Entre los años 2003 y 2006 Google publicó una serie de papers, explicando los aspectos más importantes de su infraestructura. Un año después en 2007 Amazon presentó sus ideas para su base de datos distribuida y eventualmente consistente Dynamo. Durante los años siguientes surgieron más proyectos de almacenamiento alterno, en su mayoría open-source, inspirados por Big Table y Dynamo hasta que en junio 11 de 2009 un desarrollador llamado Johan Oskarsson organizó la reunión que le daría el nombre NoSQL al movimiento [Sadalage & Fowler, 2013b]. Las bases de datos que Oskarsson convocó originalmente debían ser distribuidas, open-source y no relacionales. Hubo charlas por parte de Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB y MongoDB.

2.2.1. Características

Siendo que las bases de datos NoSQL no tienen una definición específica y cubren a toda aquella que no sea relacional, las características suelen variar entre las distintas implementaciones. Sin embargo pueden ser destacadas algunas propiedades que se mantienen en la mayoría de bases de este tipo, estas son: diseño simplificado, escalabilidad horizontal y mayor control sobre la disponibilidad de los datos. El diseño simplificado se

encuentra dado por la falta de esquemas, es decir la forma estructurada en la que se organizan los datos dentro de una base, lo cual permite implementar modelos de manera que se ajusten a las necesidades de cada sistema.
Métodos como “sharding” y “clustering” facilitan la escalabilidad horizontal, es decir, permiten separar la base de datos en distintos medios físicos, lo cual hace posible la expansión por adición de componentes y no por migración a un medio único con mayor capacidad. Esta capacidad de distribución brinda otros beneficios como por ejemplo la replicación de datos para mantener la disponibilidad sobre los mismos en caso de fallos o imprevistos. A pesar de estos beneficios resulta imposible para un sistema distribuido brindar consistencia, disponibilidad y tolerancia a particiones en simultáneo. Dicho problema es conocido como el teorema CAP [Sullivan, 2015]. Frente a esto la mayoría de bases de datos NoSQL optan por abandonar las cualidades ACID, las cuales resultan incompatibles con la idea de disponibilidad y performance de aplicaciones a escala web [MongoDB Inc, 2017b], frente a las propiedades BASE. Las propiedades BASE [Vaish, 2013] consisten en:

- Disponibilidad básica (Basic Availability): Garantiza una respuesta ante cada requerimiento, ya sea exitosa o fallida.
- Estado suave (Soft state): El estado del sistema puede cambiar con el tiempo, a veces sin ningún input previo.
- Consistencia eventual (Eventual consistency): La base de datos puede estar en un estado inconsistente momentáneamente pero eventualmente regresará a un estado consistente.

2.2.2. Tipos

Las bases de datos NoSQL suelen especializarse en problemas específicos, por ende existen distintas soluciones para distintos problemas y, en consecuencia, diferentes estrategias para la gestión de datos. Las cuatro categorías principales en las cuales se agrupan son:

- Llave/Clave valor: es el tipo más sencillo, almacenando los datos de manera completamente “libre”, sin esquema, lo cual implica que no existe una estructura definida de la información que se almacena en ella. Una llave o clave puede apuntar a un valor de cualquier tipo, desde un dato simple hasta objetos y funciones. Su principal ventaja es su facilidad de implementación y la de agregar datos. Su desventaja es la dificultad de búsqueda de elementos según su valor ya que los mismos se recuperan mediante su llave. Otra desventaja se encuentra en la dificultad de modelar relaciones entre elementos.
- Basadas en documentos: agrupan toda la información de una entidad en un documento, a su vez, los documentos pueden ser almacenados de forma conjunta en colecciones. Los documentos tienen la capacidad de almacenar subdocumentos, lo cual en RDBMS debería hacerse sobre tablas separadas o mediante un atributo especial. Los documentos en una colección pueden ser accedidos mediante una clave única.
- Basadas en columnas: a diferencia del manejo de información mediante filas de los RDBMS, este tipo de bases de datos almacenan la información en columnas. Su principal ventaja es su performance en sistemas donde se deban resolver valores máximos, mínimos, promedios y sumas sobre grandes cantidades de datos.

También se destacan en el caso de añadir nuevos valores que deban ser aplicados a todas las filas al mismo tiempo.

- ❖ Basadas en grafos: diseñadas para datos interconectados entre sí y con un número de relaciones indeterminado entre ellos. Se destacan por ejemplo para representar relaciones entre personas, como puede ser en redes sociales.

2.3. Modelo Relacional o NoSQL

Como se ha podido observar, las bases de datos NoSQL no son en sí un reemplazo para las bases de datos relacionales, sino que amplían las opciones para elegir en base a las necesidades de almacenamiento de cada aplicación en particular. Según distintos factores como el tipo y cantidad de datos, la importancia o no de transacciones precisas, la importancia de la velocidad, disponibilidad y demás, puede que sea conveniente la utilización de un modelo u otro, o la utilización de ambos para distintas partes del sistema. Por ejemplo: para una transferencia bancaria, donde son necesarias transacciones atómicas que aseguren la consistencia de las operaciones, una base de datos relacional sería la indicada.
Por otro lado, para una red social que maneja grandes cantidades de datos y usuarios relacionados entre sí, una base de datos NoSQL se ajustaría mejor a la solución. Este último ejemplo podría incluso ser implementado con dos bases de datos: una basada en documentos para el almacén de datos, en conjunto con una basada en grafos para el almacén de relaciones.

2.4. MongoDB

MongoDB [MongoDB Inc. 2017a] es una base de datos NoSQL ágil, escalable y una de las seis bases de datos que participó en la reunión de 2009, la cual popularizó el uso del nombre NoSQL para hacer referencia a aquellas no basadas en el modelo relacional. Su nombre proviene de la palabra “humongous”, cuyo significado en inglés es gigantesco. Su modelo de almacenamiento y gestión de datos se basa en documentos, modelo el cual fue descrito anteriormente en éste documento. Específicamente MongoDB utiliza el formato JSON binario o BSON para almacenar cada documento. Los objetivos principales de la base de datos son la disponibilidad, performance y escalabilidad y está orientada a páginas web de alto tráfico. Algunas de sus características principales:

- Orientada a documentos: esto significa que los datos se almacenan en la base de datos en forma de documentos, cuyo formato es similar a las estructuras de datos que se utilizarán tanto en el lado del servidor como en el cliente. Esto reduce o elimina la diferencia de impedancia que se explicó anteriormente y se puede observar con mayor detalle en el capítulo 5 de esta tesina.
- Alta performance: MongoDB es una de las bases de dato de mayor performance disponible actualmente, en especial ante tráfico pesado.
- Alta disponibilidad: el modelo de replicación de MongoDB permite mantener alta performance sin sacrificar escalabilidad.

- Alta escalabilidad: la estructura de MongoDB facilita la escalabilidad horizontal mediante el uso de sharding, es decir, la distribución de los datos sobre múltiples servidores.
- No corre el riesgo de inyecciones SQL: no es susceptible a inyecciones SQL debido a que los objetos son almacenados como objetos y no mediante cadenas de caracteres SQL.

2.5. API REST

Es usual encontrar que sistemas de dominios y especificaciones diferentes, tengan la necesidad de interoperar. Es decir, establecer algunos medios de interacción máquina-máquina (o proceso-proceso o directamente entre programas o sistemas), basados en un protocolo de comunicación compartido y prepactado, para cumplir con objetivos diversos, como pueden ser:

- Compartir datos (o acceder a información no disponible localmente)
- Reutilización de componentes compartidos
- Actualización de estados de ciertos elementos
- Utilización de funcionalidades remotas o el tratamiento de funciones de callback entre distintos sistemas, que den aviso de la ocurrencia de algún evento específico (creación de un nuevo usuario, cambio de estado de una cuenta, finalización de la ejecución de un proceso, etc).

En este marco, existe una tecnología que ha venido ganando terreno entre los distintos recursos de interoperabilidad a los que se recurre para comunicar sistemas de forma automática, conocida como REST. De sus iniciales en inglés (REpresentational State Transfer), REST es un estilo de arquitectura basado en un conjunto de principios que describe cómo son definidos y referenciados recursos sobre la red. Estos principios fueron descritos por primera vez en 2000 por Roy Fielding como parte de su disertación doctoral [Fielding, R. 2000]. Es importante tener en cuenta que REST no se encuentra definido en un documento formal que especifique explícitamente sus propiedades como puede ser un RFC (Request for Comments), lo cual implica que la implementación entre cada sistema puede variar considerablemente. No obstante a lo anteriormente mencionado existen 6 principios definidos en el documento de Fielding que se consideran los requisitos básicos que debe cumplir una aplicación para poder ser considerada REST se mencionan a continuación:

1. Arquitectura Cliente–servidor

Consiste en separar la lógica del cliente (interfaz de usuario) de la del servidor (almacenamiento de datos), lo cual mejora la portabilidad de la interfaz de usuario sobre múltiples plataformas. Esto a su vez significa que los componentes pueden evolucionar de manera independiente, permitiendo el soporte de los requerimientos a escala web de múltiples dominios organizacionales.

2. Sin estado

La comunicación entre el servidor y el cliente se realiza sin que se almacenen datos del contexto del cliente dentro del servidor entre cada requerimiento al mismo. Esto significa que cada requisito proveniente de cualquier cliente contiene toda la información necesaria para ser atendida, guardándose el estado en cliente. El estado de la sesión puede ser transferido por el servidor a otro servicio, como puede ser una base de datos, para mantener el estado persistente por un periodo y permitir autenticación. Cuando el cliente se encuentre listo para realizar una transición a otro estado puede comenzar a enviar requerimientos nuevamente. El proceso será considerado en estado de transición mientras uno o más requisitos hayan sido enviados y se encuentren a la espera de una respuesta.

3. Uso de caché

Como su funcionamiento en la World Wide Web, clientes e intermediarios pueden almacenar sus respuestas en caché. Las mismas deben definir implicita o explicitamente si permiten o no dicha funcionalidad, previniendo de esta manera que clientes obtengan información errónea en respuesta a futuros requerimientos. El uso de caché elimina parcial o completamente la necesidad de algunas interacciones entre el cliente y servidor, aportando así performance y escalabilidad.

4. Sistema en capas

Un cliente por lo general no tiene forma de saber si su conexión es directamente con el servidor final o un intermediario. Servidores intermediarios pueden aportar a la escalabilidad del sistema al permitir el balanceo de carga y proveer caches compartidas. El uso de los mismos permite además reforzar políticas de seguridad.

5. Código bajo demanda (opcional)

Los servidores pueden extender o personalizar temporalmente la funcionalidad de un cliente transfiriendo código ejecutable. Ejemplos de esto pueden incluir componentes compilados, como los applets de Java y los scripts del lado del cliente, como JavaScript.

6. Interfaz uniforme

Una interfaz uniforme es fundamental para el diseño de cualquier servicio REST. Esto simplifica y permite desacoplar la arquitectura, permitiéndole a cada parte evolucionar independientemente. A continuación se listan los cuatro requisitos que constituyen la interfaz uniforme:

6.a) Identificación de recursos en el requisito: Los recursos individuales se identifican en las solicitudes, por ejemplo, utilizando URI en sistemas REST basados en la web. Los recursos mismos se encuentran conceptualmente separados de las representaciones que le son devueltas al cliente. Por ejemplo, el servidor puede enviar datos de su base de datos como HTML, XML o JSON, ninguno de los cuales es la representación interna del servidor.

6.b) Manipulación de recursos a través de representaciones: Cuando un cliente tiene una representación de un recurso, incluidos los metadatos adjuntos, tiene suficiente información para modificar o eliminar el recurso.

6.c) Mensajes autodescriptivos: Cada mensaje incluye suficiente información para describir cómo procesar el mensaje. Por ejemplo, el parseador a invocar puede ser especificado por un tipo de medio.

6.d) Hipermedia como motor del estado de la aplicación (HATEOAS): Al acceder a una URI inicial para la aplicación REST -análogo a un usuario web humano que accede a la página de inicio de un sitio web- un cliente REST debería poder utilizar enlaces provistos por el servidor de forma dinámica para descubrir todas las acciones y recursos disponibles que necesita. A medida que avanza el acceso, el servidor responde con texto que incluye hipervínculos a otras acciones que están actualmente disponibles. No es necesario que en el cliente se encuentre especificada la información sobre la estructura o dinámica del servicio REST.

2.6. Aplicaciones REST y API REST

Las aplicaciones construidas siguiendo esta arquitectura son a veces denominadas aplicaciones REST, y en general son orientadas a la web, como mínimo como implementaciones de sistemas distribuidos en principio siguiendo la estructura general cliente/servidor. Las API REST son generalmente utilizadas para permitir que distintos sistemas (denominados sistemas cliente o solicitante) puedan acceder y manipular representaciones textuales de recursos web, a partir de la utilización de un conjunto predefinido y uniforme de operaciones sin estado (normalmente utilizando los comandos del protocolo HTTP: GET, POST, PUT o DELETE a través de Internet). Como toda API, permite la inter-operación entre aplicaciones, y al utilizar el protocolo HTTP se hace accesible/se tiene acceso en la web en general.
Tal y como se define en [Ramanathan & Raja, 2013] “REST ignora los detalles de la implementación de los componentes y la sintaxis del protocolo, con el objetivo de hacer foco en los roles de estos componentes, las restricciones sobre sus interacciones con otros y su interpretación de los elementos de datos significativos”. Por esto, la implementación de una API REST es el desarrollo de un sistema que cumpla con los lineamientos definidos por este estándar, sin importar la tecnología o el dominio circundantes. Como ejemplo de esto, se pueden enumerar diversos tipos de REST APIS implementadas y en funcionamiento, que fueron construidas en tecnologías diferentes, que cumplen cometidos variados y son diariamente consultadas por incontables sistemas (REST Countries [Florez, 2018], Accuweather [AccuWeather Inc, 2018], Fixer.io [Fixer.io, 2018], Cloud Convert [Lunaweb ltd, 2018], Twitter [Twitter, Inc, 2018], Google Drive [Google LLC, 2018]).
En este contexto, este trabajo plantea la construcción de una API REST utilizando tecnologías sobre las cuales se propuso realizar pruebas de rendimiento, esto permitió, entre otras cosas: estudiar la implementación del protocolo REST utilizando herramientas de última generación, obtener un sistema construido por capas completamente funcional

para poder realizar las pruebas antes mencionadas, estudiar la carga que genera la utilización de cada capa y lo que esta aporta, para establecer conclusiones sobre el costo-beneficio de la inclusión o no de la misma en el sistema, analizar otras cuestiones asociadas al funcionamiento (tolerancia a fallos, consistencia, validación de datos, transacciones, etc) y, finalmente, poder generar conclusiones sobre cómo reacciona la herramienta a diferentes volúmenes de datos, analizando el rendimiento de cada capa de forma individual y su influencia sobre el conjunto.

Luego de haber establecido estos conceptos básicos sobre las bases de datos NoSQL e introducido a MongoDB, en el capítulo siguiente serán definidas las estructuras utilizadas para la realización de las pruebas propuestas en el capítulo 1. Además para ese mismo fin será definida la plataforma a utilizar, explicando cada componente o “capa” que la conforma.

Capítulo 3 | Capas de Abstracción de la Aplicación

En este capítulo será definida la estructura de datos a utilizar para la realización de las pruebas de rendimiento y se explicará en detalle la infraestructura de software que da soporte a una aplicación de bases de datos NoSQL con MongoDB como motor de la/s base/s de datos. Se verá en detalle cómo se encuentra compuesta la arquitectura de la aplicación, definiendo cada uno de sus componentes o “capas” de software en el lado del servidor, desde el propio MongoDB como DBMS hasta la interfaz REST para la operación con la/s aplicación/es.

.

3.1. Estructura/Ejemplo a Utilizar

El modelo datos dado en la figura 3-1 es el utilizado para los experimentos de evaluación de rendimiento. Aunque simple, este modelo nos es útil para verificar distintos aspectos de interés, como las inserciones múltiples de documentos interrelacionados. Para automatizar la creación de elementos a insertar en la base de datos, se crearon una serie de scripts encargados de realizar dicha tarea sobre cada componente que conforma la arquitectura de la aplicación. Dicha arquitectura, junto con los componentes que la conforman serán detallados a continuación en este capítulo.

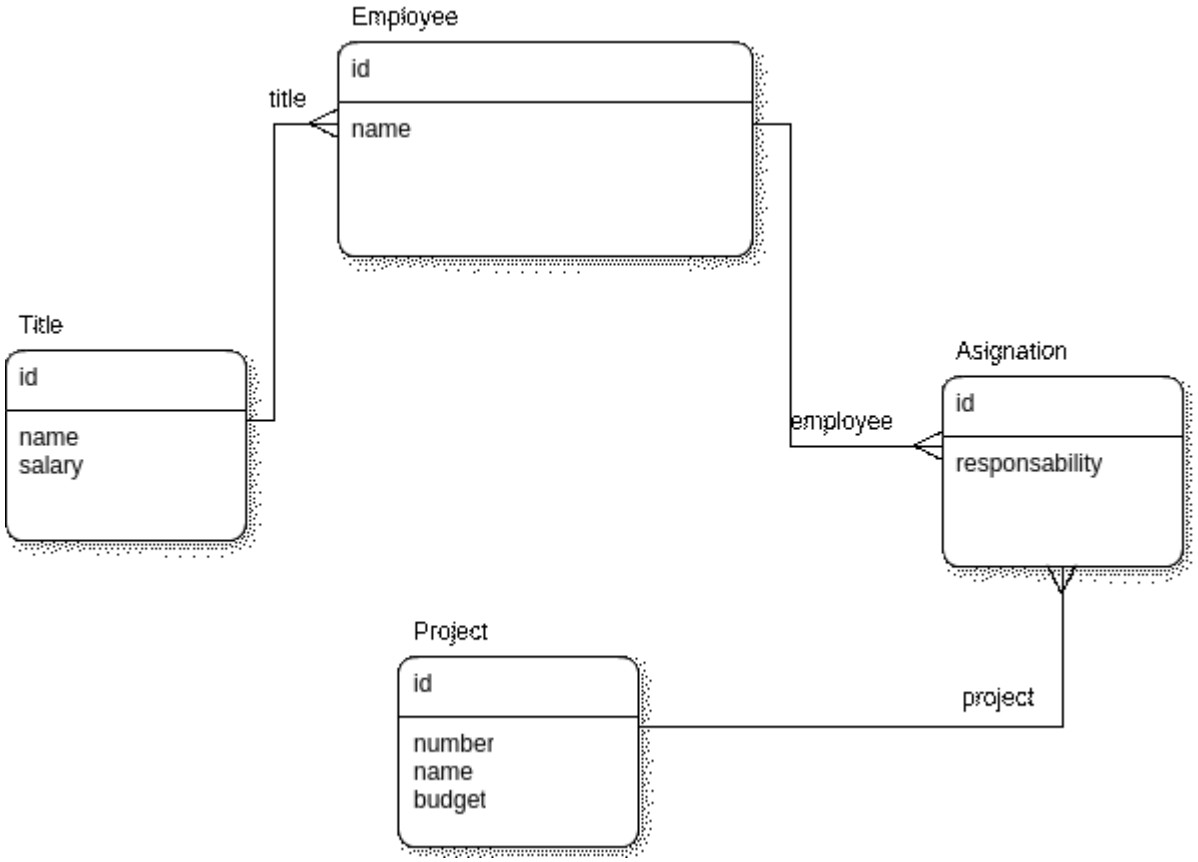

Figura 3-1:Diagrama UML del modelo de datos utilizado para las pruebas.

Estos scripts de automatización de inserciones mencionados anteriormente asignan los mismos datos de tamaño fijo para cada elemento que generan (a excepción de los identificadores únicos de cada elemento), asegurándose así que no haya un desperdicio innecesario de tiempo de generación de datos aleatorios y, al ser los datos de tamaño fijo, permiten calcular con facilidad el tamaño total de datos a insertar.
Para manipular el tamaño de los datos utilizando el modelo de la figura 3-1 se utiliza alguno de los campos de tipo string presente en cada colección y se le asignan caracteres aleatorios. Conociendo el tamaño de cada caracter (o *char/character*, un string en realidad es una secuencia de caracteres) se puede dividir el tamaño deseado de dato por este tamaño de caracter para calcular la cantidad de caracteres necesarios. Este proceso se verá con detalle en el capítulo 6 de este documento.

3.2. Arquitectura de la Aplicación

En la práctica, cuando se crea una aplicación web, no solo es necesario un motor de bases de datos sino también un conjunto de subsistemas de software que en conjunto forman una plataforma, lo que se conoce como "software stack" o pila de software [Techopedia Inc, 2018]. El propósito de esta plataforma es facilitar la construcción y el mantenimiento de la aplicación a crear sin la necesidad de agregar software adicional. En el caso de MongoDB, la plataforma más popular es la que se conoce como "MEAN stack" [Mean.io, 2018] cuyo apodo deriva de las iniciales de los componentes que la conforman, los cuales son:

- MongoDB: manejador de base de datos orientado a documentos multiplataforma gratuito y de código abierto
- Express: un framework web minimalista que opera del lado del servidor y fue escrito para Node.js
- AngularJS: framework web para front-end (lado de usuario), ejecuta código JavaScript en el navegador para implementar interfaces de usuario "reactivas", es decir que se redibujan dinámicamente según el input del usuario.
- Node.js: ambiente o entorno de ejecución asincrónico y basado en eventos para la ejecución de código JavaScript del lado del servidor.

El propósito de gran parte de este trabajo es el análisis de la posible sobrecarga de trabajo y su efecto en el rendimiento de cada elemento o "capa" que conforma la plataforma en el lado del servidor. Esto permitirá analizar concretamente el trabajo extra que se genera con cada elemento o "capa" adicional, en caso de que exista. Para dicho fin, AngularJS será excluido del estudio, siendo que es un framework basado en la lógica del lado del cliente.
Por otro lado, se incluirá Mongoose, un ODM (Object Document Mapper) que define los modelos a utilizar para almacenar los datos en forma de documentos. Esta capa es particularmente útil ya que permite darle una estructura a los datos de MongoDB, los cuales originalmente son libres de esquema, permitiendo la validación automática en la inserción o actualización de documentos.
Cabe aclarar que el término "capa" es especialmente adecuado para referirse a un elemento dentro de la plataforma, debido a que cada elemento suma a la funcionalidad inicial de

MongoDB, por lo cual de ahora en adelante utilizaremos el término con dicho fin. Con el objetivo de medir esta diferencia en performance, se realizaron un conjunto de pruebas de inserción de datos. Las mismas fueron realizadas sobre las distintas capas de la plataforma, analizando en cada caso su rendimiento ante una incremental cantidad de datos de entrada. Luego de haber establecido las capas a analizar: MongoDB, Mongoose y Express tal como se las muestra esquemáticamente en la figura 3-2, se analizará a continuación el funcionamiento o rol que cumple cada una. Cabe destacar que Node.js no se considera una de las capas en sí, siendo éste uno de los posibles ambientes de ejecución de MongoDB (lo que se conoce como un driver) más que una capa de la plataforma.

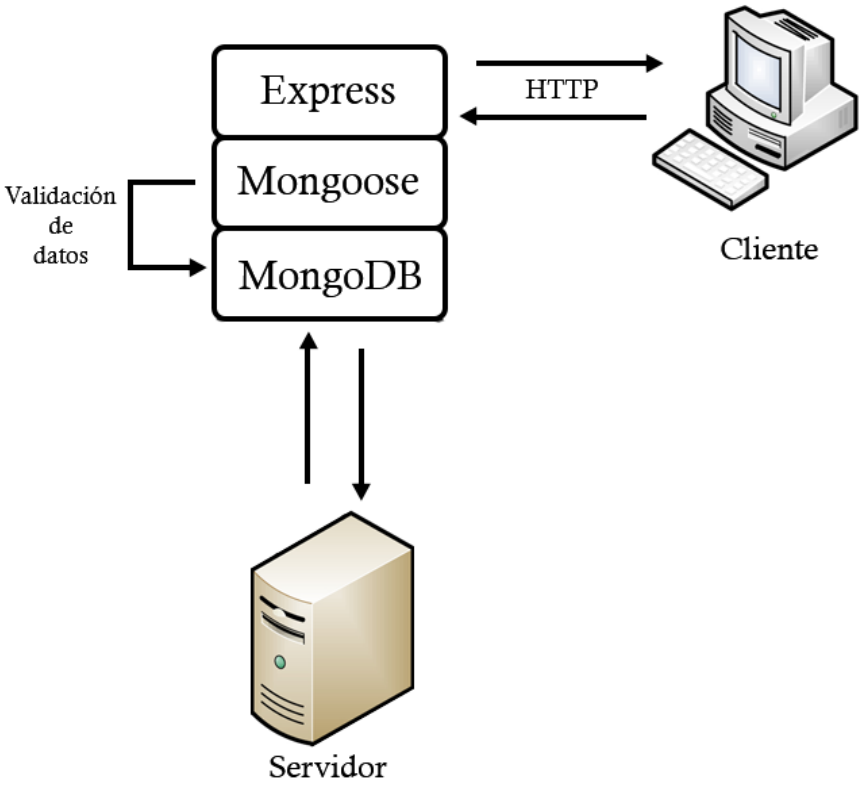

Figura 3-2: Representación gráfica de la pila de software utilizada para las pruebas en funcionamiento.

3.3. Capa 1 - MongoDB

MongoDB actúa como la capa base y su funcionalidad es, como se explicó a lo largo del capítulo 2 de este documento, la de una base de datos, permitiendo almacenar datos y posteriormente acceder a los mismos. Particularmente para el caso de MongoDB, el almacenamiento de datos se realiza en forma de documentos BSON, una estructura JSON de JavaScript que luego se almacena en código binario. Los drivers de MongoDB, en el caso de este trabajo Node.js, envían y reciben datos en BSON.
En el caso de la estructura de datos a utilizar para las pruebas y modelada anteriormente en la figura 3-1, por ejemplo, serán necesarias cuatro colecciones (una por cada una de las

clases planteadas). Para obtener una figura más clara de cómo se almacena un documento, se muestra a continuación un elemento de la colección “Assignation” de la forma:

```
{
        "_id" : ObjectId("000000000000000000000001"),
        “responsibility” : “Una responsabilidad”,
        “employee” : ObjectId("000000000000000000000001"),
        “project” : ObjectId("000000000000000000000001")
}
```

El atributo “_id” representa el identificador único de cada documento de la colección, todo objeto en MongoDB posee uno propio. Por otro lado el atributo “responsibility” es un campo propio de la colección “Assignation” de tipo string. Finalmente los atributos “employee” y “project” son referencias a documentos pertenecientes a las colecciones de su mismo nombre, por lo cual estos campos almacenan una referencia a dichos documentos.

3.3.1. Funcionamiento de MongoDB

Una instalación típica de MongoDB consta de dos elementos principales:

1) Por un lado “mongo”, una interfaz shell interactiva que proporciona un ambiente para que administradores de sistemas o los desarrolladores mismos puedan interactuar directamente sobre la base de datos, permitiendo así probar consultas o “queries” y operaciones sobre la misma. Esta interfaz provee un ambiente JavaScript completamente funcional para su uso con MongoDB.
2) Por otro lado se encuentra “mongod” que representa el proceso principal de MongoDB, el cual se ejecuta como un daemon. Este proceso maneja los pedidos de datos, administra el acceso de los mismos y realiza operaciones de gestión que se ejecutan de fondo.

Por defecto “mongod” permanece corriendo de fondo, escuchando en el puerto 27017 por pedidos entrantes. Normalmente solo se ejecuta una instancia de “mongod” sobre un servidor particular, mientras que múltiples instancias del proceso “mongo” (interfaz/shell) pueden ser ejecutadas en un mismo momento.

3.4. Capa 2 - Mongoose

Mongoose es considerada la segunda capa a analizar en este trabajo, su función principal es la de validar los datos que son almacenados en la base de datos. Como ya se mencionó anteriormente en este capítulo, Mongoose es un ORM para Node.js que proporciona una solución directa basada en esquemas para modelar los datos de la aplicación. Incluye casting de tipo, validaciones, construcción de consultas, hooks de lógica de negocios y demás. Estas validaciones que realiza Mongoose sobre los documentos de la base son definidos mediante modelos conocidos como “Schemas” o esquema. Al tener un modelo de

datos definido, la aplicación sabe qué campos y tipos de valor esperar para un para un documento de una determinada colección. Siguiendo el ejemplo presentado para el caso anterior, a continuación se muestra el modelo o esquema definido para la colección "Assignation":

```
{
//importar los módulos necesarios
        var mongoose = require('mongoose');
        var Schema = mongoose.Schema;
        var ObjectId = mongoose.Schema.Types.ObjectId;


// definición del esquema
var AssignationSchema = new Schema({
        _id: ObjectId,
        responsibility: String,
        project_id: [{type: Schema.ObjectId, ref: "Project"}],
        employee_id: [{type: Schema.ObjectId, ref: "Employee"}]
});

module.exports = mongoose.model('Assignation', AssignationSchema);
}
```

Por medio de la utilización del modelo definido anteriormente, Mongoose realiza una serie de validaciones al momento de realizar algún tipo de operación sobre los datos de la colección en cuestión, incluyendo la inserción de un nuevo elemento en la misma. La utilización de estos esquemas para el caso de una inserción implica principalmente que:

- Cualquier campo que se intente agregar que no pertenezca al modelo será ignorado.
- Un elemento con campos faltantes puede ser incorporado, de hecho pueden ser establecidos en el modelo valores por defecto.
- Se evitará cualquier intento de inserción donde el tipo de dato no coincida con el especificado en el esquema. Por ejemplo la asignación de un número al campo responsibility provocará un error por intento de conversión implícito de tipo número a string.

Con estos nuevos controles y restricciones que Mongoose agrega sobre MongoDB se gana robustez sobre el modelo. Por otro lado, a cambio de dicha robustez se pierde en cierta medida la flexibilidad propia de las bases NoSQL y puede llegar a dar una impresión de semejanza con respecto al comportamiento de una base relacional. Este es, sin embargo, solo un parecido y a fines prácticos al momento de realizar modificaciones al modelo se observa como diferencia que:

- En Mongoose: la agregación o eliminación de campos solo implica modificar el esquema definido. En el caso de la eliminación de un atributo ningún documento almacenado en la base previo al cambio perderá el atributo a menos que se especifique explícitamente.

- En una base relacional: cualquier tipo de modificación implica un cambio en la tabla entera. La eliminación de un atributo puede no ser posible dependiendo la base de datos utilizada, generando la necesidad de una migración de la tabla con todos sus elementos. Incluso si la base permite la eliminación de columnas, esta se realiza de manera retroactiva, lo cual quiere decir que no existe forma de que los datos insertados previos a la eliminación de la columna conserven el atributo.

Mongoose además presenta un conjunto extenso de funcionalidades adicionales que no fueron necesariamente requeridas para las implementaciones planteadas en este trabajo. Algunas de las más importantes son:

- La posibilidad de guardar métodos de clase o estáticos y métodos de instancia dentro de los modelos. La diferencia entre ambos es que los estáticos se ejecutan sobre el modelo y los de instancia sobre un documento de mongoose (instancia del modelo). Los métodos estáticos funcionan de manera similar a un stored procedure de SQL.
- Propiedades virtuales: Mongoose permite definir campos virtuales para un documento de manera que puedan ser accedidos y modificados pero no persisten al momento de almacenar el documento en la base de datos.
- La posibilidad de escuchar todo pedido de inserción o actualización que ingresa en la base de datos, siempre y cuando se trabaje con replicación.
- Población de campos: Esta funcionalidad permite reemplazar automáticamente dentro de un documento la referencia almacenada a un documento de otra colección por el objeto completo con sus respectivos campos.
- Discriminadores: son un mecanismo de herencia de esquema que permite definir distintos modelos y asignarlos a una misma colección de la base de datos.
- Plugins: el uso de plugins permite extender la funcionalidad inicial de Mongoose con el uso de funciones pre-empaquetadas en forma de plugins, ya sea por el mismo usuario o por la comunidad.
- Middleware: la posibilidad de agregar o modificar las funciones que realizan las validaciones para personalizarlas. Esto es particularmente conveniente ya que permite extender la funcionalidad inicial de mongoose, permitiendo crear hooks o “ganchos” que se ejecuten antes, durante o después de una operación en particular, lo cual brinda un nivel mayor de control sobre la base de datos subyacente.

3.5. Capa 3 - Express

Express.js es un framework para Node.js diseñado para la construcción de aplicaciones web y APIs y es considerado como la tercera y última capa a utilizar en este trabajo. Esta capa es la encargada de brindar conectividad con procesos externos mediante el uso del repertorio de operaciones HTTP que proporciona Express.js. Mediante el uso de Express se definió y estructuró además la api REST siguiendo los conceptos básicos definidos en el capítulo anterior. La implementación particular de api REST llevada a cabo en este trabajo se encuentra estructurado de la forma planteada en la figura 3-3.

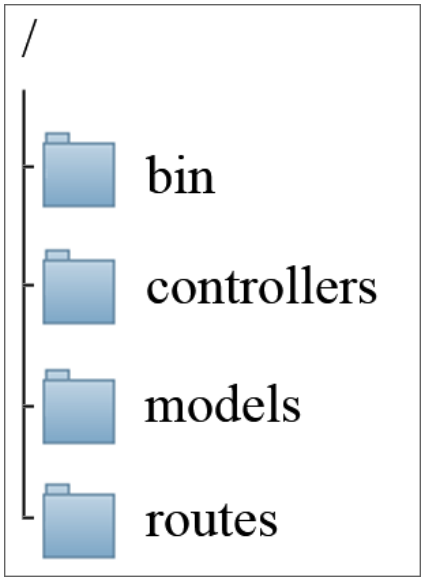

Figura 3-3, estructura del sistema de archivos que conforman la plataforma propuesta.

En la figura 3-3 se pueden observar las cuatro carpetas principales del sistema.

- Bin: en esta carpeta se almacenan los scripts principales necesarios para medir los tiempos de cada capa. Estos se analizarán con mayor detalle en capítulos posteriores.
- Controllers: En esta carpeta se almacenan los controladores, los cuales poseen la funcionalidad necesaria para realizar las inserciones de la capa 3. Esto es distinto a las capas 1 y 2, donde toda la funcionalidad necesaria para realizar las inserciones de elementos se encuentra incluida en los scripts correspondientes de la carpeta “bin”.
- Models: Aquí se encuentran los archivos conteniendo los esquemas o modelos de cada colección a utilizar por Mongoose a partir de la capa 2 y como se vio en uno de los ejemplos planteados anteriormente en este capítulo.
- Routes: En esta carpeta se almacenan los archivos que contienen las rutas permitidas para cada una de las colecciones.

Para tener una visión más clara de cómo se define cada una de estas rutas, las cuales representan las operaciones permitidas para cada colección, se analizará a continuación los contenidos del archivo de rutas para la colección “Assignation”:

```
// importar los módulos necesarios
var assigController = require('../controllers/assignationController');
var express = require('express');
var router = express.Router();

// api/assignations/
router.route('/api/assignations/')
    .post(assigController.createAssigMany);
    .get(assigController.listAssignations);

module.exports = router;
```

En el ejemplo anterior se observa cómo se puede acceder y operar sobre un recurso web (en este caso el recurso “Assignation”) a partir de la utilización de un conjunto predefinido y uniforme de operaciones sin estado (HTTP), tal y como se analizó en el capítulo anterior. Para definir las rutas se utilizan métodos del objeto “app” de Express que se corresponden con los métodos HTTP del mismo nombre, en el caso del ejemplo se observan los métodos para POST y GET.
Los métodos de ruteo especifican una función de callback (a veces referida como función handler) llamada en el momento que la aplicación recibe un pedido dirigido a la ruta especificada. En el caso del ejemplo anterior, cualquier pedido post realizado a la ruta “/api/assignations/” será atendido en la función “createAssigMany” y cualquier pedido por get será atendido en la función “listAssignations” del controlador “assignationController”.

3.6. Tecnología Relacionada a Cada Capa

Como se explicó a lo largo de este capítulo, cada capa de la plataforma planteada cumple un rol distinto y, a su vez, se encuentra relacionado a un ambiente de ejecución distinto, asociada a su propia tecnología en particular. Esto se puede ver más claramente mediante la figura 3-4.

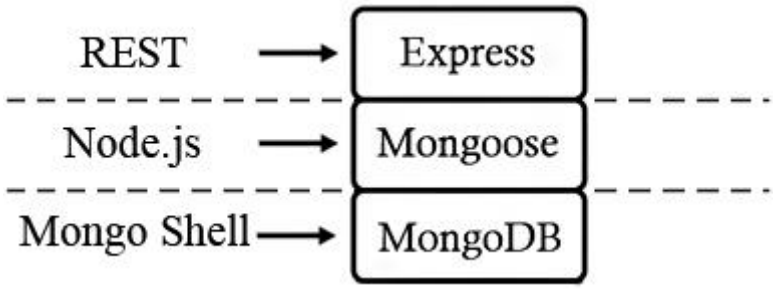

Figura 3-4: tecnología empleada en cada capa de la plataforma.

En el caso de la primera capa: MongoDB la ejecución de código se realiza mediante Mongo Shell, la consola interactiva propia de MongoDB. Esta consola se ejecuta sobre el motor de

JavaScript SpiderMonkey, actualmente sostenido por la fundación Mozilla. SpiderMonkey implementa el estándar ECMA-262 edición 5.1 con algunas funcionalidades adicionales y es el reemplazo del motor V8 a partir de la versión 3.2 de MongoDB.
Por otro lado, en la segunda capa: Mongoose se ejecuta sobre Node.js, luego el código se traduce en operaciones sobre la base de datos mediante el driver de mongo que le corresponde, en este caso el driver Node.js. Node.js utiliza un motor de JavaScript V8 actualizado, el cual implementa las especificaciones del estándar ECMA-262.
En la tercera capa: Express es utilizado como el framework web para Node.js que implementa las interfaces necesarias mediante HTTP para gestionar los requerimientos a la aplicación. Además, como la aplicación fue modelada como una API REST se tiene por un lado el código del servidor, encargado de escuchar pedidos y establecer las configuraciones necesarias, y por otro lado el código con la funcionalidad propia de la aplicación. Esta funcionalidad se encuentra dividida en controladores que responden a las operaciones sobre cada colección. A su vez, los controladores se comunican con los modelos, definidos como esquemas de Mongoose, los cuales se conectan a la base de datos mediante el driver de Node.js ante una operación sobre los datos.
Una vez definidos los modelos y la tecnología a utilizar y luego de haber establecido las capas en cuestión, se definirá en el capítulo siguiente las configuraciones necesarias previo a la realización de las pruebas específicas de evaluación de rendimiento más extensivas.

Capítulo 4 | Configuración Inicial y Precondiciones

Antes de realizar las pruebas de rendimiento para cada capa de la plataforma es necesario determinar la configuración considerara “óptima” o “mejor” para MongoDB. Aunque es difícil determinar a priori una configuración óptima absoluta (y válida para todas las situaciones o escenarios que se puedan presentar en las aplicaciones), es necesario como mínimo identificar combinaciones entre el tamaño de los datos y tamaño de buffer para la cual la ejecución de las inserciones se completada de forma exitosa y en el menor tiempo posible.
Este estudio inicial nos permitirá, además, documentar algunas características propias de MongoDB, como algunos métodos de inserción, siendo que MongoDB presenta varias alternativas posibles para la inserción de uno o varios elementos en la base de datos. Es por esto que se detalla a continuación el análisis realizado mediante una serie de pruebas con el fin de identificar los mejores valores encontrados para las diferentes alternativas que conforman la configuración.

Todos los experimentos se llevaron a cabo en una computadora con las características de hardware y software que se listan a continuación:

- Sistema operativo: Linux Mint 18 Cinnamon 64 bit.
- Procesador: i5 3570k / 2 nucleos 3.40GHz.
- RAM: 4Gb 667MHz.
- Disco rigido: 400 Gb libres.

4.1. Métodos de Inserción

MongoDB presenta varias alternativas cuando se trata de insertar datos en la base, a continuación se define cada una junto con los resultados que obtuvieron en las pruebas: Inserción Directa, Bulk Ordenado y Bulk Desordenado. En todos los casos, se tiende a insertar varios datos/documentos a la vez, dado que el foco está en la inserción de datos en la base de datos MongoDB.
Inserción Directa se refiere a que los datos fueron almacenados en arreglos o vectores del tamaño indicado por un “Tamaño buffer” (específicamente la cantidad de elementos del vector) y luego insertados en la base de datos sin ninguna instrucción adicional. Los otros dos métodos de inserción son por **“Bulk” o lote de instrucciones** [MongoDB Inc, 2017f], en esta técnica las instrucciones de inserción (no los datos mismos) son almacenados en un lote del tamaño indicado por un “Tamaño buffer”, cuando éste se llena, las instrucciones son ejecutadas. En este caso, el valor de “Tamaño buffer” es de configuración para MongoDB, para que el propio sistema sea el que controla cuándo efectivamente se almacenarán los datos. Se ha utilizado el mismo nombre que en el caso de la Inserción Directa porque se refieren conceptualmente a lo mismo: cantidad de elementos a insertar. Para el caso de **Bulk Ordenado**, la ejecución de las instrucciones se realiza de manera secuencial y según el orden original de las mismas, una instrucción debe esperar a que la anterior termine para poder comenzar a ejecutarse. En el caso del **Bulk Desordenado** en cambio, las instrucciones pueden ser ejecutadas en paralelo, siendo éste en teoría más rápido, dicha

mejora debería hacerse más visible en una base de datos distribuida. En el caso de la Inserción Directa utilizando un vector como en nuestro caso, se tendría una variante “programada” de la Inserción Bulk Ordenado. En los otros dos casos, sería el propio MongoDB el que efectivamente insertará de acuerdo a la configuración que se le haya indicado en términos de “Tamaño Buffer” y Bulk Ordenado o Bulk Desordenado.En todos los casos, la cantidad de elementos a insertar es la misma (parámetro mencionado como”Tamaño buffer”), solo cambia la forma en que los documentos se insertan de manera efectiva en la base de datos.

4.2. Evaluación Experimental de los Métodos de Inserción

En las tablas se pueden observar los resultados de la columna “Tiempo”, que indica el tiempo en segundos que tardó la inserción total de los datos de una prueba particular. Se utilizará además la columna “Posición” para resaltar los resultados, según el rendimiento obtenido entre las 3 técnicas.

4.2.1. Inserción Directa

En la Tabla 4-1 se muestran los resultados de tiempos de necesarios para diferentes alternativas de la inserción que se ha denominado “Inserción Directa”, es decir insertar un vector de una determinada cantidad de elementos. Cada columna de la Tabla 4-1 indica un valor en particular:

- La columna “Tamaño Buffer” es la que muestra la cantidad de elementos del vector insertado, es decir con tamaños de vector de entre 16 y 256 elementos.
- La columna “Tamaño elemento” es la cantidad de Megabytes de cada elemento en particular, que en este caso varía entre 1Mb y 5Mb según la prueba.
- La columna “Cantidad Elementos” indica la cantidad total de elementos insertados, recordando que se insertaron en grupos de “Tamaño Buffer”.
- La columna “Total Insertado” muestra la cantidad de bytes insertados, que es exactamente la cantidad total de elementos insertados multiplicada por el tamaño de cada elemento.
- La columna “Posición” representa el valor relativo de una prueba en particular con respecto a los 3 métodos de inserción testeados. Con este fin el valor 1 representa el menor tiempo obtenido entre las 3 pruebas realizadas bajo los mismos parámetros y distinto método de inserción.

De esta manera, la primera fila de la Tabla 4-1 muestra el tiempo requerido para insertar 1024 elementos de 1MB cada uno, agrupados o insertados utilizando un vector de 16 elementos. Dicho de otro modo: se necesitaron poco más de 16s para insertar 64 (1024/16) vectores de 16 elementos cada uno, quedando la colección con 1024 documentos más. De manera análoga, la segunda fila muestra un leve incremento del tiempo necesario para insertar la misma cantidad de documentos, pero en este caso agrupados en vectores de 32 elementos. Teniendo en cuenta las primeras 5 filas de la Tabla 4-1, se puede ver que agrupar más documentos a insertar implica un crecimiento en el total del tiempo necesario

para que MongoDB lleve a cabo las inserciones. Las últimas 5 filas de la Tabla 4-1 muestran algo similar cuando el total de datos a insertar es de 5GB y se agrupan también en vectores de 16 a 256 elementos. Sin embargo, en términos de tiempo por elemento o por GB insertar mayor cantidad de datos muestra un crecimiento de tiempo que es algo mayor al esperable lineal. En el mejor de los casos, almacenar 1GB de datos de 1MB c/u requiere 37.178s mientras que almacenar 5GB de datos de 1MB c/u es bastante mayor a cinco veces lo anterior, dado que en el mejor de los casos se requieren 269.616s, es decir algo más de 7.25 veces el tiempo de 1GB (el crecimiento lineal "esperable" sería de 5 veces más). En todos los casos, esta forma de insertar documentos en la base de datos es la peor entre las tres probadas, y es por eso que en la columna "Posición" aparece en todos los casos el número 3. En los resultados que siguen, no solamente se podrá notar que se obtienen mejores tiempos de inserción con los otros dos métodos sino que se podrá hacer la comparación cuantitativa de la diferencia.

Tamaño Buffer	Tamaño Elemento	Cantidad Elementos	Total Insertado	Tiempo (s)	Posición
16	1MB	1024	1GB	37.178	3
32	1MB	1024	1GB	37.923	3
64	1MB	1024	1GB	37.836	3
128	1MB	1024	1GB	38.781	3
256	1MB	1024	1GB	41.372	3
16	1MB	5120	5GB	269.616	3
32	1MB	5120	5GB	307.672	3
64	1MB	5120	5GB	288.572	2
128	1MB	5120	5GB	293.83	2
256	1MB	5120	5GB	310.043	2

Tabla 4-1: tiempos de Inserción Directa de Datos.

4.2.2. Bulk Ordenado

Tal como se comentó anteriormente, el Bulk Ordenado es una posibilidad de inserción de MongoDB que se puede aprovechar en los casos en los cuales se tengan que insertar varios/muchos documentos en una colección de una base de datos MongoDB. Se utiliza este método configurando el tamaño/cantidad de documentos a insertar, que se ha denominado "Tamaño Buffer" y para el cual se

han utilizado los mismos valores que en el caso anterior de “Inserción Directa”. La Tabla 4-2 muestra los resultados obtenidos, que comparativamente son casi todos mejores que en el caso anterior a excepción de los últimos tres valores (5GB en total de datos insertados, con “bulks” de 64, 128 y 256 elementos/documentos, respectivamente). Aunque estos últimos tres valores absolutos de tiempo son peores que en el caso de inserción directa, la diferencia relativa no es tan significativa, dado que en el peor de los casos es de menos del 20% de tiempo. A excepción de esos tres últimos casos de inserción de muchos datos y agrupados en “bulks” relativamente grandes, este método es en general mejor que el de Inserción Directa y peor que el método Bulk Desordenado. Los dos casos en que es mejor que el Bulk Desordenado (segunda y cuarta fila de la Tabla 4-2, con valores de Posición “1”) no son significativamente mejores, tal como se verá a continuación.

Tamaño Buffer	Tamaño Elemento	Cantidad elementos	Total insertado	Tiempo (s)	Posición
16	1MB	1024	1GB	33.678	2
32	1MB	1024	1GB	33.979	1
64	1MB	1024	1GB	36.457	2
128	1MB	1024	1GB	36.196	1
256	1MB	1024	1GB	36.625	2
16	1MB	5120	5GB	246.496	2
32	1MB	5120	5GB	250.464	2
64	1MB	5120	5GB	308.66	3
128	1MB	5120	5GB	342.897	3
256	1MB	5120	5GB	348.85	3

Tabla 4-2: Tiempos de Inserción por Bulk Ordenado de Datos.

4.2.3. Bulk Desordenado

La Tabla 4-3 muestra los resultados de inserciones con el método de Bulk Desordenado para los mismos casos/experimentos/datos que se mostraron anteriormente. Como en cierta manera se podría esperar, al tener MongoDB mayores grados de libertad en cuanto a cuándo llevar a cabo la inserción de cada elemento o documento en particular, se tienen mejores resultados de tiempo. Se debe recordar que tanto en la Inserción Directa como en

la de Bulk Ordenado los documentos se insertan en el orden dado por el programador, y en el caso del Bulk Desordenado es MongoDB el que decide cuándo insertar cada documento en particular, independientemente del orden en el cual se insertaron en tiempo de ejecución del cliente particular. Los dos casos en los cuales el Bulk Desordenado aparece como "peor" que el Bulk Ordenado, que se muestran en la segunda y cuarta filas de la Tabla 4-3 con el número "2" en realidad son peores en valores absolutos de tiempo, pero las diferencias no son significativas respecto a los mejores valores absolutos: 33.979 vs. 34.069 y 36.196 vs 37.532 segundos respectivamente.

Tamaño Buffer	Tamaño Elemento	Cantidad elementos	Total insertado	Tiempo (s)	Posición
16	1MB	1024	1GB	31,949	1
32	1MB	1024	1GB	34.069	2
64	1MB	1024	1GB	32.302	1
128	1MB	1024	1GB	37.532	2
256	1MB	1024	1GB	35.82	1
16	1MB	5120	5GB	231.106	1
32	1MB	5120	5GB	238.256	1
64	1MB	5120	5GB	255.084	1
128	1MB	5120	5GB	247.025	1
256	1MB	5120	5GB	265.078	1

Tabla 4-3: Tiempos de Inserción por Bulk Desordenado de Datos.

4.3. Configuración "Ideal"

A partir de los resultados preliminares que se mostraron en la sección anterior, el método de inserción Bulk Desordenado obtuvo los mejores resultados de rendimiento y en algunos casos por un gran margen de diferencia. También es notable cómo en primera instancia el método de Bulk Ordenado parece casi tan eficiente como el desordenado pero ante una entrada de datos mayor el rendimiento no se mantiene, e incluso llega a ser peor que el de la inserción directa de datos, que en general se mantiene como el de peor rendimiento en la mayoría de los casos.
Luego, se puede apreciar además como para cada uno de los métodos, el mejor de sus tiempos siempre se encuentra dado por la entrada con el buffer de tamaño 16 (elementos), tanto para las pruebas de 1GB como para las de 5GB de datos en total. Con el objetivo de

verificar algunas variantes de tamaño de cada elemento las pruebas siguientes fueron realizadas con dicho tamaño de buffer y utilizando el método de inserción por Bulk Desordenado. Con estos experimentos ya se elegiría la combinación más eficiente entre todos los parámetros: cantidad de elementos, tamaño de “bulk”, cantidad total de datos y tamaño individual de cada dato. La Tabla 4-4 muestra los tiempos de inserción requeridos para diferentes tamaños de cada documento a insertar, manteniendo la agrupación en “bulks” (Tamaño Buffer) de 16 elementos y para cantidades totales de 1GB y 5GB de datos.

Tamaño Buffer	Tamaño Elemento	Cantidad elementos	Total insertado	Tiempo (s)
16	16Kb	65536	1GB	33.858
16	32Kb	32768	1GB	38.029
16	64Kb	16384	1GB	36.194
16	128Kb	8192	1GB	37.147
16	256Kb	4096	1GB	36.261
16	512Kb	2048	1GB	34.957
16	16Kb	65536	5GB	204.112
16	32Kb	32768	5GB	207.329
16	64Kb	16384	5GB	197.059
16	128Kb	8192	5GB	192.505
16	256Kb	4096	5GB	292.721
16	512Kb	2048	5GB	262.752

Tabla 4-4: Tiempos por Bulk Desordenado y Diferentes Tamaños de Elementos.

En esta nueva prueba utilizando la configuración descrita anteriormente (Tamaño Buffer 16 y Bulk Desordenado) se buscó aquel tamaño con menor tiempo de ejecución en cada caso. Los mejores tiempos estuvieron dados por los tamaños 16Kb para inserciones de 1GB de datos y 128Kb para las inserciones de 5GB. Como se puede observar para pruebas con muestras no tan grandes, un tamaño de elemento de 16Kb es ideal siendo el más rápido. Elementos más grandes son comparativamente más lentos, sin embargo a partir de cierto punto (alrededor de los 2GB o más), comienza a perder su eficacia. Por otro lado, el tamaño de elemento 128Kb comienza siendo uno de los más lentos para tamaños pequeños y evoluciona en el más rápido para inserciones mayores a 3GB. Cabe destacar que tamaños mayores a 128Kb no solo son peores sino que hacen más lenta la ejecución en algunos

casos a más del 50%. Por estas razones se utilizó un tamaño de 128Kb para las pruebas de rendimiento.

4.4. Errores/Problemas y Limitaciones Encontrados

Durante la realización de estas pruebas preliminares se encontraron algunos aspectos configurables de la plataforma y de las herramientas utilizadas (ej.: tiempos de timeout) que facilitan y optimizan la realización de las pruebas. Muchos de estos aspectos específicos de la implementación serán tratados en el capítulo 6, donde se analizan los aspectos técnicos, particulares del desarrollo de los algoritmos de inserción. No obstante esto, también se encontró un error particular de la plataforma de trabajo MongoDB. Este problema es un error que conlleva a que los requerimientos de memoria principal sean mayores a la que el sistema operativo le puede asignar al proceso que inserta elementos, generando así la terminación del mismo. El error se produce debido a que no se puede liberar la suficiente memoria a tiempo entre una inserción y otra. El proceso continúa realizando requerimientos de asignación de más memoria hasta que alcanza el límite y el sistema operativo interrumpe el proceso. Esto ocurre incluso al utilizar las herramientas que brinda el mismo MongoDB, es decir los bulk de inserciones, los cuales justamente tienen el propósito de insertar elementos en cantidad.

Inicialmente se pensó que el problema de excesiva utilización de memoria encontrado se debía a un error en la configuración por defecto de MongoDB, lo cual podría generar una mala administración en el uso de memoria. La variable de configuración encargada de la asignación de memoria en MongoDB es “wiredTigerCacheSizeGB” a la cual se le debería asignar por defecto el valor mayor entre el 50% de (RAM - 1 GB), o 256 MB, es decir máx((RAM - 1GB)/2, 256MB) [MongoDB Inc, 2017e]. Sin embargo este es un límite impuesto sobre “mongod”, el proceso daemon principal de MongoDB, lo cual no asegura que exista un límite sobre las instancias de mongo shell o sobre los drivers. Esto implicaría que en la práctica, en una instalación estándar, siguiendo todos los pasos indicados por la documentación que corresponde, no hay ningún límite (no al menos conocido a priori) para los procesos.

En la Figura 4-1 se muestra un ejemplo de la ejecución de una de las pruebas que es finalizado forzosamente por el sistema operativo. Esta prueba en particular fue realizada con un tamaño de datos de 512Kb y un tamaño de buffer de 64 elementos.

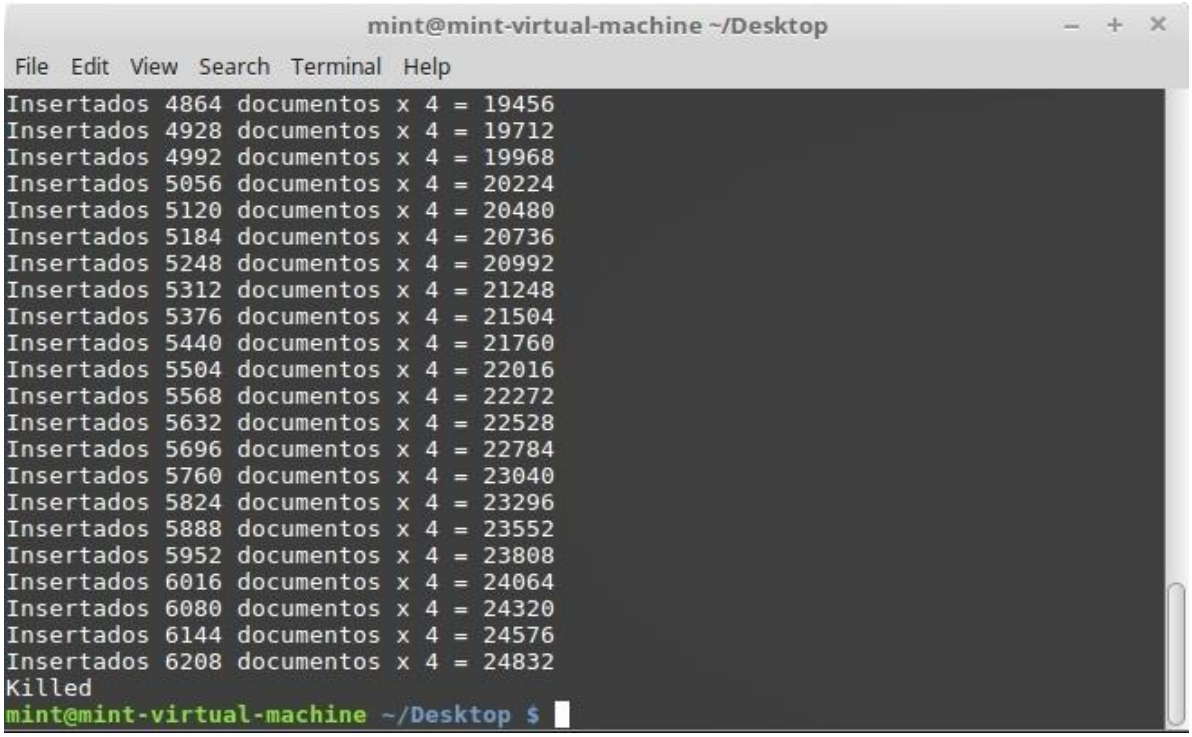

Figura 4-1: Error por falta de memoria durante una inserción de 100GB de datos.

En el ejemplo anterior (mostrado en la Figura 4-1), puede verse la falla en la ejecución luego de haberse alcanzado los 24832 elementos (24832 x 512Kb = 12,1 GB) para la configuración propuesta. La Figura 4-2 muestra la monitorización de la ejecución del proceso de inserción utilizando otra consola/terminal con el comando top..

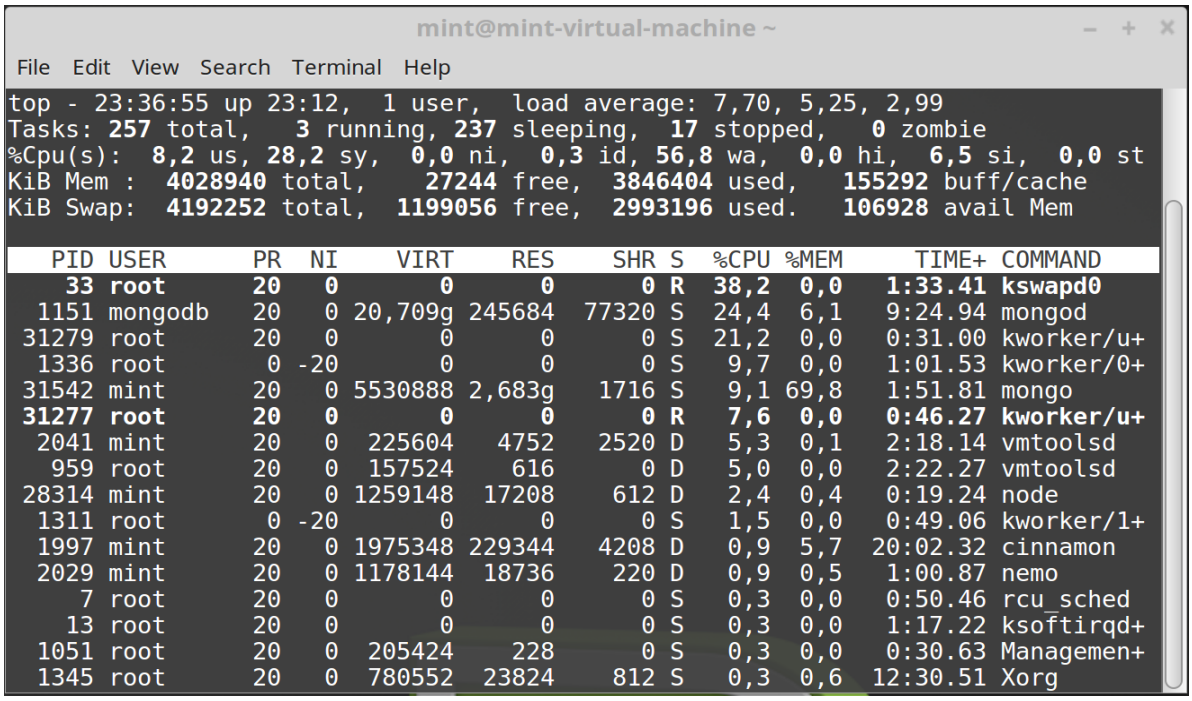

Figura 4-2: Monitorización (comando top) de la inserción que falla.

En la figura 4-2, en la que se muestra la salida del comando top, se puede verificar cómo el proceso “mongo” (proceso de consola/shell de MongoDB) consume casi un 70% de memoria, con “mongod” (proceso propio del DBMS MongoDB) consumiendo otro 6% adicional. Considerando que el sistema cuenta con 4GB de memoria principal, un 70% de uso se traduce en aproximadamente 2,8GB. Este valor es un 86,6% (o 1,3GB) más alto del que el máximo que podría llegar a utilizar el proceso “mongod” considerando el tamaño dictado por “wiredTigerCacheSizeGB”, el cual equivale a 1,5GB.

Por último, cabe destacar que el proceso daemon principal de MongoDB, “mongod”, no es interrumpido cuando falla la ejecución de “mongo”. Sumado al hecho de que ejecutando la inserción por medio de Node.js se obtiene el mismo resultado que con mongo shell, esto lleva a la conclusión de que el problema probablemente radica en la naturaleza misma de JavaScript. La conclusión más probable es que ante cada pedido asincrónico de inserción de JavaScript se genera un callback, los cuales a su vez se apilan más rápidamente de lo que pueden ser resueltos, finalizando en la saturación de la memoria. Se podría probar la conclusión anterior mediante la realización de la misma serie de pruebas sobre MongoDB pero haciendo uso de algún otro conector distinto de Node.js (Ruby, Python, etc) para ver si realmente esta es la causa del problema.

Todo lo anterior llevó a la necesidad de implementar una combinación de la cantidad de datos insertados en un mismo momento y el tamaño de dichos datos para lograr un balance entre la memoria que se tiene ocupada por el proceso y la que se libera. Esta configuración ideal es la que se describió anteriormente en este capítulo, pero es importante tener en cuenta que dependiendo de las características de la máquina utilizada, esta configuración puede ser distinta. Es posible que configuraciones que no funcionen en una máquina determinada puedan ser ejecutadas en otra con mayor cantidad de memoria principal o mejor relación de tiempo de ejecución entre el subsistema de E/S y memoria sin problemas, siendo que se tiene más espacio o posibilidades para llegar al punto crítico entre la memoria utilizada y la liberada.

Capítulo 5 | Análisis de los Resultados Obtenidos

En este capítulo se describirán distintos experimentos y los resultados de rendimiento, enfatizando los factores sobre los cuales se realizaron las pruebas de rendimiento. Asimismo, se analizarán los resultados obtenidos capa por capa, para cada una de las detalladas en el capítulo anterior en las distintas pruebas, para finalmente comparar estos resultados entre sí y establecer algunas conclusiones fundamentadas.

5.1. Consideraciones Iniciales

Una vez establecida la configuración necesaria se procedió a realizar las pruebas. Dichas pruebas consisten en ejecutar una serie de scripts diseñados para generar datos de entrada, insertarlos en la base de datos, medir el tiempo que lleva almacenarlos en su totalidad y finalmente verificar la integridad de los mismos. La necesidad adicional de verificar la consistencia de los elementos que se insertan en la base de datos viene dada en parte por las características particulares de MongoDB y para asegurar la correctitud de las pruebas y por ende las mediciones que se obtengan.
En MongoDB una operación sobre un único documento es atómica. Debido a la posibilidad de usar documentos embebidos para representar relaciones entre datos en una sola estructura de documento en lugar de normalizar sobre múltiples documentos y colecciones, esta atomicidad de documento único evita la necesidad de transacciones de varios documentos para muchos casos de uso práctico. Sin embargo, cuando una operación de escritura modifica múltiples documentos al mismo tiempo, por ejemplo mediante una instrucción updateMany(), a pesar de que la modificación de cada documento individual es atómica, la operación como un todo no lo es. Esto último quiere decir que si la ejecución de una operación de este tipo es interrumpida, todo elemento insertado conservará su integridad pero no se asegura que todos los elementos sean insertados.
De todas formas, en el intento de proceder de la forma más meticulosa posible, la verificación de consistencia también incluyó un chequeo de integridad de todos los documentos insertados para asegurar la validez de cada prueba de rendimiento. Cada uno de los algoritmos que fueron utilizados para realizar las pruebas que figuran en el transcurso de este capítulo serán analizados con mayor detalle en el capítulo siguiente.

5.2. Pruebas de Rendimiento

Mediante el uso del modelo descrito en el capítulo 3 y en combinación con la configuración establecida en el capítulo 4, se realizaron inserciones en la base de datos a través de cada uno de los niveles de la plataforma. Cabe aclarar también que los componentes que forman el modelo de datos se encuentran relacionados entre sí, como una medida que ayude a distinguir si las validaciones de la segunda capa sobre dichas relaciones presenta algún impacto en su rendimiento.
Para realizar los experimentos a partir de cuyos resultados se evaluará el rendimiento, se tuvieron en cuenta distintos factores, entre ellos:

- Tamaño de dato: cantidad de bytes que ocupa cada elemento individual.
- Cantidad de elementos: la cantidad de elementos a insertar en la prueba.
- Tamaño total de prueba: la suma total del tamaño de todos los elementos.
- Tamaño de buffer: la cantidad de elementos a insertar en un mismo momento, los mismos se almacenan en una misma estructura de datos para luego ser almacenados en la base de datos por medio del método de Bulk Desordenado.
- Tiempo de ejecución: el tiempo que tomó la operación en completar la inserción de todos los elementos de una misma prueba.
- Verificación de consistencia: el tiempo que tomó completar la verificación de los datos previamente insertados.

Estos factores se traducen en los distintos campos de las tablas de resultados que se muestran más adelante en este capítulo. Estos factores se podrían clasificar como de configuración, indicadores, variables, de resultado y adicionales. Los factores de configuración serían Tamaño de dato y Tamaño de buffer, que hacen referencia a aquellos campos y/u opciones que permanecen constantes para todas las pruebas de una capa, su valor debió ser determinado antes de la ejecución de las mismas ya que el mismo repercute directamente sobre los resultados (ver capítulo 4). Un campo indicador es aquel que no representa ni un “input” ni un “output” de la prueba, sino que son simplemente datos derivados de otros campos que presentan información adicional relevante, por ejemplo la columna “Cantidad de elementos” puede obtenerse a partir de la división entre el tamaño total de la prueba y el tamaño de dato. Un campo variable es el de Tamaño total de prueba, es el campo cuyos valores difieren para cada prueba y que, junto con la configuración, cumplen el rol de “input” de las mismas. Los campos de resultados corresponden a los tiempos de ejecución, son aquellos campos que representan los datos obtenidos de cada prueba y cumplen el rol de “output” de una prueba. Sus valores dependerán de la combinación entre los campos de configuración y los variables y son el foco de estudio de este trabajo. Adicionalmente, se encuentra el campo verificación de consistencia el cual es un resultado que depende del tamaño total de la prueba y es, a su vez, independiente de la configuración. A continuación se describen las pruebas realizadas sobre cada capa de la plataforma y los resultados obtenidos junto con el análisis de los mismos.

5.2.1. Capa 1: MongoDB

Para el primer conjunto de pruebas fueron testeados los distintos métodos de inserción a la base de datos, teniendo en cuenta también su rendimiento para los distintos tamaños de buffer (cantidad de elementos insertados en un mismo momento). Luego de este análisis, el cual fue presentado en el capítulo anterior, se optó por un tamaño de buffer de 16 elementos de 128Kb cada uno con inserción mediante bulk desordenado como la configuración a utilizar. La Tabla 5-1 muestra los valores absolutos de tiempos de ejecución obtenidos en las pruebas de rendimiento para la primer capa, la misma se encuentra dividida en las columnas que se definieron en la sección anterior. Por otro lado la Figura 5-1 muestra estos mismos datos representados de forma gráfica. Para mayor claridad el gráfico fue separado en las Figuras 5-1 y 5-2, mostrando su comportamiento en las pruebas de

pequeña y mediana cantidad de datos y en las de grandes cantidades de datos, respectivamente.

Prueba de rendimiento:

Tamaño Buffer	Tamaño Elemento	Cantidad elementos	Total insertado	Tiempo Inserción (s)	Verificación de consistencia (s)	Tiempo Total (s)
16	128Kb	8	1MB	0,195	0,021	0,216
16	128Kb	80	10MB	0,735	0,087	0,822
16	128Kb	800	100MB	5,851	3,283	9,134
16	128Kb	8192	1GB	37,147	33,773	70,92
16	128Kb	16384	2GB	76,985	67,197	144,182
16	128Kb	24576	3GB	126,031	102,008	228,039
16	128Kb	32768	4GB	162,791	133,715	296,506
16	128Kb	40960	5GB	192,505	164,321	356,826
16	128Kb	49152	6GB	255,317	208,979	464,296
16	128Kb	57344	7GB	298,744	238,113	536,857
16	128Kb	65536	8GB	338,653	274,006	612,659
16	128Kb	73728	9GB	350,259	297,648	647,907
16	128Kb	81920	10GB	380,393	339,886	720,279
16	128Kb	163840	20GB	779,952	662,894	1442,846
16	128Kb	409600	50GB	2.035,415	1696,689	3732,104
16	128Kb	819200	100GB	3.998,877	3335,329	7334,206

Tabla 5-1: Resultados obtenidos en las pruebas de rendimiento para la primer capa.

Como es posible observar en las Figuras 5-1 y 5-2, más allá de una ejecución comparativamente lenta para los primeros valores debido a la desventaja que presenta el uso de elementos de 128Kb en inserciones pequeñas (como se estableció en las pruebas de configuración), el tiempo de ejecución de las inserciones es relativamente lineal. Hasta este punto no hay mucho más para analizar, ni se podría inferir ninguna anomalía en los

tiempos de los experimentos, dado que es lo que a priori se esperaba: operaciones con mayor cantidad de datos implican un tiempo proporcionalmente mayor (lineal) de las operaciones. Lo que sí es importante es que estos tiempos van a ser considerados los "tiempo de referencia" para comparar/evaluar la sobrecarga de las capas adicionales de software que se utilizarán a continuación.

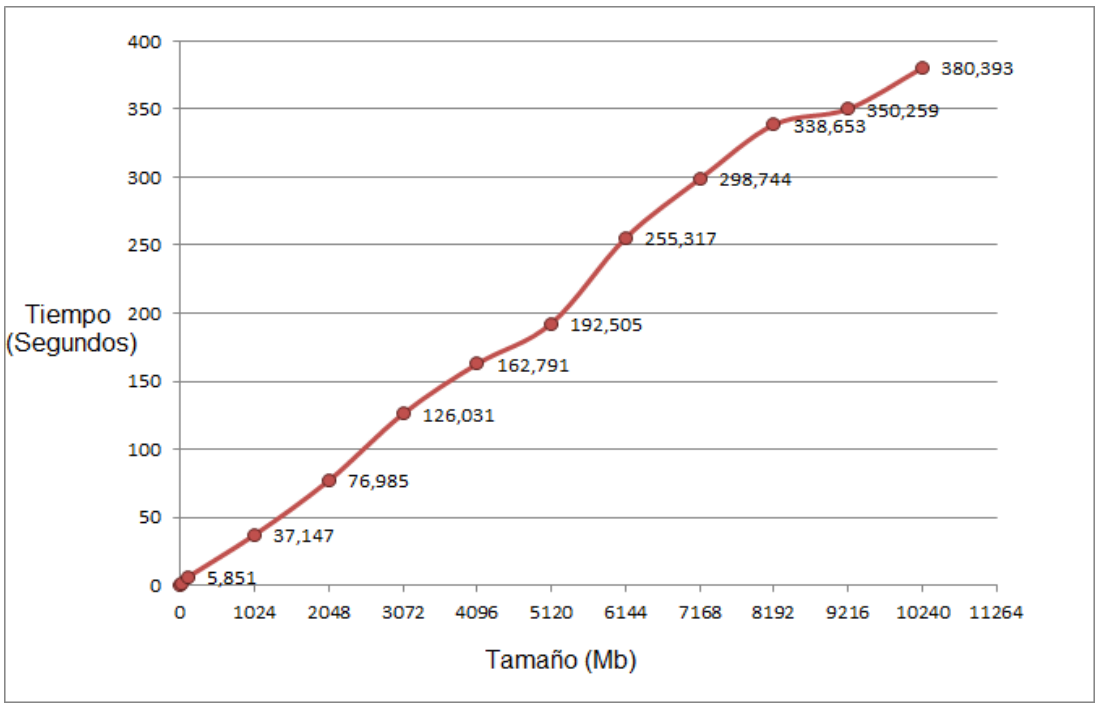

Figura 5-1: Resultados obtenidos de las pruebas realizadas a la capa 1 (hasta 10 GB).

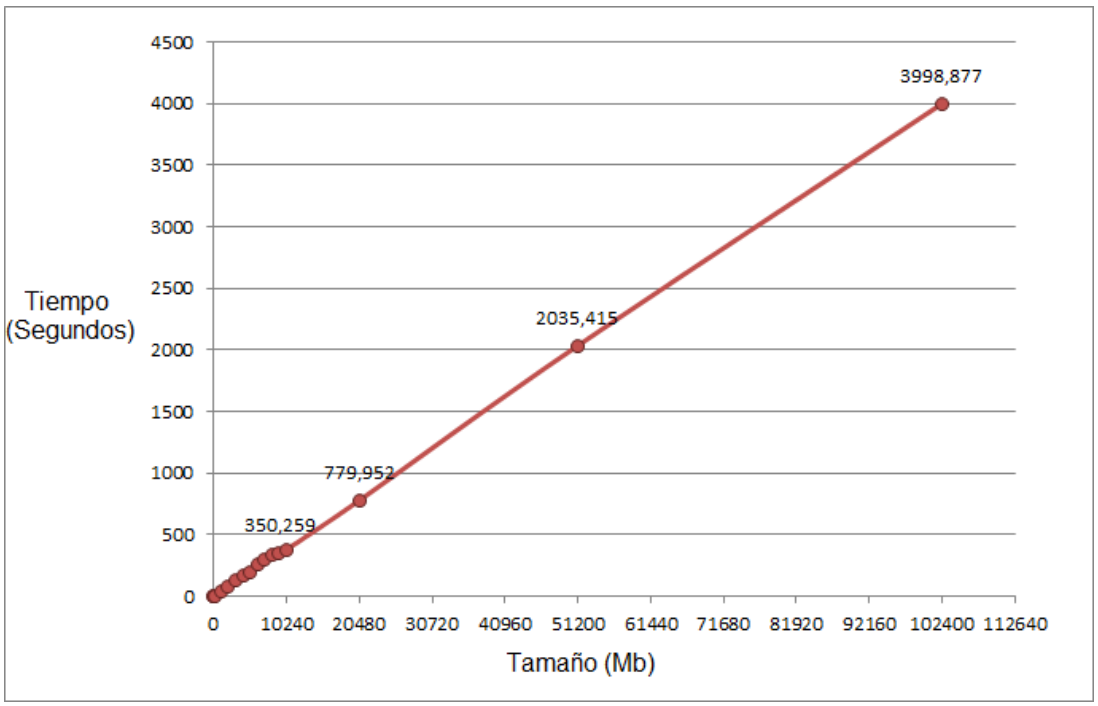

Figura 5-2: Resultados obtenidos de las pruebas realizadas a la capa 1 (hasta 100 GB).

5.2.2. Capa 2: Mongoose

Para medir la diferencia en cada capa es necesario que las pruebas de rendimiento se realicen con la configuración más parecida posible (lo más parecido que permita cada implementación) para que la comparación sea justificable [IBM, 2018b] y la diferencia representada sea el overhead real que exista o no entre dichas capas. Para dicho fin, el tamaño de buffer y elemento serán los mismos que para la capa 1 (utilizados en los experimentos anteriores). Sin embargo, se utilizará como técnica de inserción de datos el comando “create” propio de Mongoose en lugar de bulk, el cual sobrepasa los controles y validaciones, comunicándose directamente con el driver de MongoDB.
Para evitar errores en las pruebas y la necesidad de re-inserción de elementos fallidos, fue necesario reconfigurar la conexión a la base de datos para permitir ignorar los tiempos de timeout. Se debe tener en cuenta que esto presenta un caso ideal que no suele ocurrir en la práctica, pero a los fines de este documento es necesario para medir de forma correcta cada caso de prueba.
En la Tabla 5-2 se muestran los valores absolutos de tiempos de ejecución obtenidos para las pruebas realizadas y, de la misma forma que para las pruebas de la primera capa, las Figura 5-3 y 5-4 muestran estos mismos datos representados de forma gráfica. Nuevamente se observa [Figuras 5-3 y 5-4] un crecimiento lineal proporcional a la cantidad de datos a insertar en la base, lo cual en sí mismo ya es satisfactorio, dado que indica que la capa Mongoose no impone una sobrecarga diferente para diferentes cantidades de datos. Además, los resultados obtenidos en términos de valores absolutos fueron muy similares a los de la capa anterior, con tiempos en algunos casos levemente menores. La conclusión sobre la posible razón de dicha mejora será tratada con más detalle en la comparación final de los resultados de cada capa.

Tamaño Buffer	Tamaño Elemento	Cantidad elementos	Total insertado	Tiempo (s)	Verificación de consistencia (s)	Tiempo Total (s)
16	128Kb	8	1MB	0,128	0,021	0,149
16	128Kb	80	10MB	0,184	0,087	0,271
16	128Kb	800	100MB	1,656	3,283	4,939
16	128Kb	8192	1GB	34,218	33,773	67,991
16	128Kb	16384	2GB	75,458	67,197	142,655
16	128Kb	24576	3GB	118,26	102,008	220,268
16	128Kb	32768	4GB	156,93	133,715	290,645
16	128Kb	40960	5GB	198,758	164,321	363,079
16	128Kb	49152	6GB	236,329	208,979	445,308
16	128Kb	57344	7GB	279,843	238,113	517,956
16	128Kb	65536	8GB	311,811	274,006	585,817
16	128Kb	73728	9GB	362,134	297,648	659,782
16	128Kb	81920	10GB	381,925	339,886	721,811
16	128Kb	163840	20GB	777,318	662,894	1440,212
16	128Kb	409600	50GB	1.960,016	1696,69	3656,705
16	128Kb	819200	100GB	4.128,604	3335,33	7463,933

Tabla 5-2: Resultados obtenidos en las pruebas de rendimiento para la segunda capa.

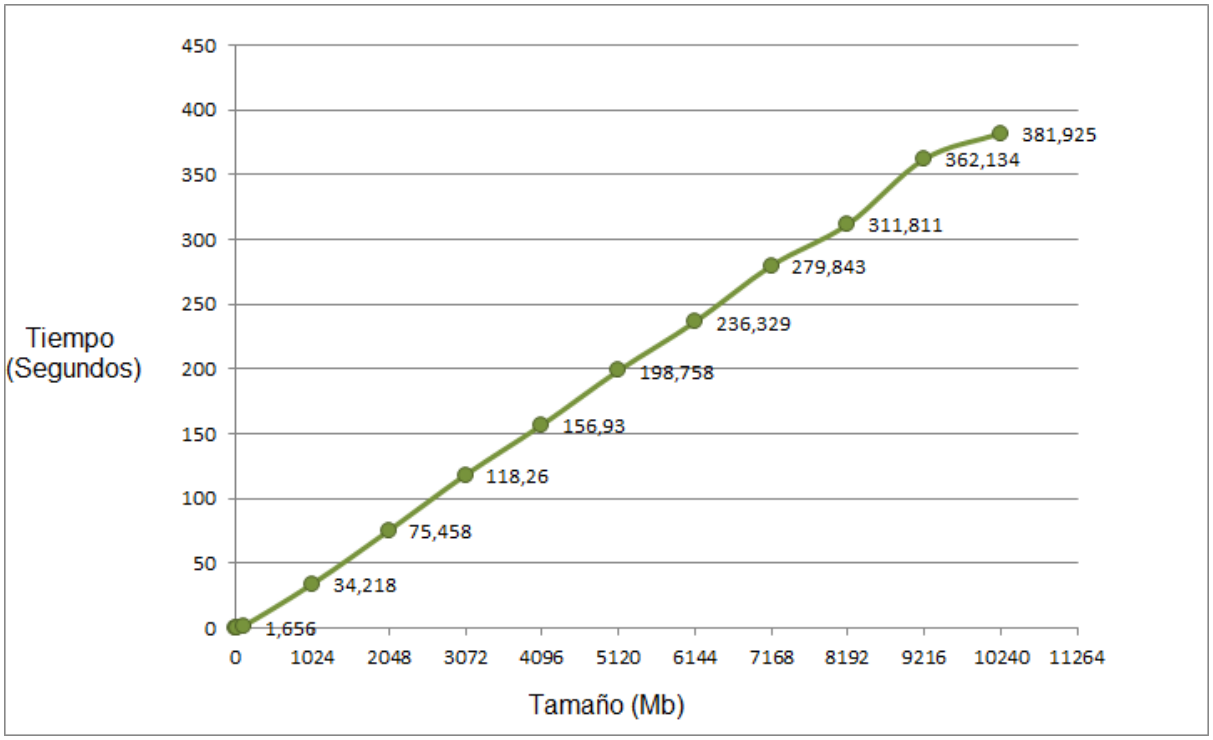

Figura 5-3: Resultados obtenidos de las pruebas realizadas a la capa 2 (hasta 10 GB).

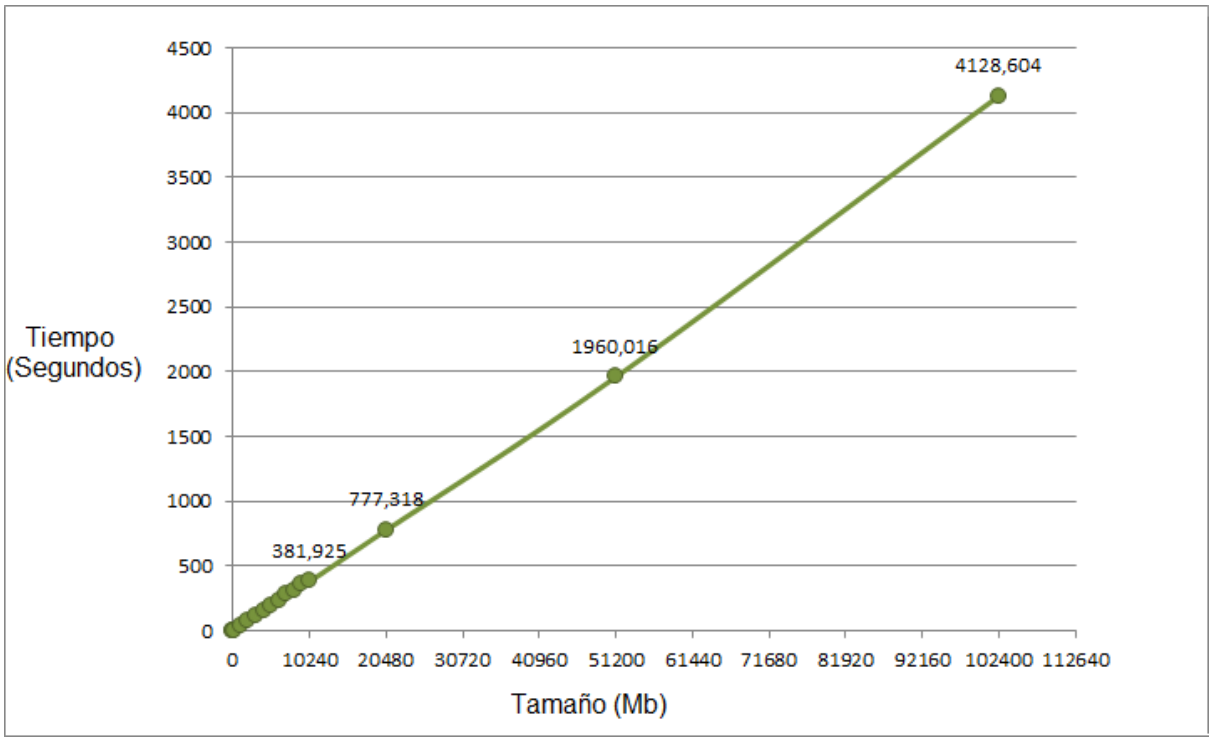

Figura 5-4: Resultados obtenidos de las pruebas realizadas a la capa 2 (hasta 100 GB).

5.2.3. Capa 3: Express

Al igual que para la capa anterior, se buscó que las pruebas de rendimiento se realicen con la configuración más parecida posible. Sin embargo, la naturaleza misma de la tercera capa hace que sea necesario tomar en consideración algunas características adicionales. Específicamente lo que se intentó medir en esta capa es el tiempo adicional generado por la comunicación y transferencia de datos entre procesos mediante el protocolo HTTP. Dichos procesos actúan como un cliente que realiza pedidos de inserción y un servidor que realiza la inserción real sobre la base de datos. Bajo estas circunstancias, en la Tabla 5-3 los valores de la columna “Tamaño buffer” representan ahora la cantidad de datos enviados en un mismo request mediante el protocolo HTTP.
Por otra parte, del lado del servidor las inserciones aún se agrupan y ejecutan mediante el comando “create” de Mongoose. Cabe destacar en esta capa que fue necesaria la inclusión de la librería sync-request para node.js, la cual obliga la espera de la respuesta del servidor ante cada request antes de enviar la petición siguiente. Sin dicha librería el programa cliente no puede manejar la cantidad de operaciones request necesarias en cada prueba y es terminado por falta de memoria. Aún así, pruebas realizadas con requests de distintos tamaños revelaron que no se encontró una diferencia sustancial entre los tiempos de ejecución, lo cual sugiere que los resultados obtenidos en las pruebas que se mostrarán a continuación representan el tiempo real de transferencia de datos mediante el protocolo HTTP y no un overhead de tiempo de comunicación entre procesos a causa de la secuencialidad de los mensajes.
Como último detalle y, de forma similar a lo ocurrido en la segunda capa con respecto a reconfiguraciones necesarias para la ejecución de las pruebas, vale tener en cuenta que fue necesario cambiar especificaciones de los requisitos HTTP para que se acepten comunicaciones de hasta 50MB de transferencia de datos, lo cual en la práctica es poco común. En la Tabla 5-3 se muestran los valores absolutos de tiempos de ejecución obtenidos para las pruebas realizadas y ,de la misma forma que para las pruebas de la primer y segunda capa, las Figura 5-5 y 5-6 muestran estos mismos datos representados de forma gráfica. Una vez más se observa en las Figuras 5-5 y 5-6 un crecimiento lineal, sin embargo la ejecución de las inserciones en esta capa es notablemente más lenta.

Tamaño Buffer	Tamaño Elemento	Cantidad elementos	Total insertado	Tiempo (s)	Verificación de consistencia (s)	Tiempo Total (s)
16	128Kb	8	1MB	0,539	0,021	0,56
16	128Kb	80	10MB	0,975	0,087	1,062
16	128Kb	800	100MB	4,107	3,283	7,39
16	128Kb	8192	1GB	40,443	33,773	74,216
16	128Kb	16384	2GB	88,391	67,197	155,588
16	128Kb	24576	3GB	136,723	102,008	238,731
16	128Kb	32768	4GB	182,755	133,715	316,47
16	128Kb	40960	5GB	235,688	164,321	400,009
16	128Kb	49152	6GB	281,251	208,979	490,23
16	128Kb	57344	7GB	326,983	238,113	565,096
16	128Kb	65536	8GB	374,361	274,006	648,367
16	128Kb	73728	9GB	424,433	297,648	722,081
16	128Kb	81920	10GB	482,625	339,886	822,511
16	128Kb	163840	20GB	921,113	662,894	1584,007
16	128Kb	409600	50GB	2.334,522	1696,69	4031,211
16	128Kb	819200	100GB	4.634,82	3335,33	7970,146

Tabla 5-3: Resultados obtenidos en las pruebas de rendimiento para la tercer capa.

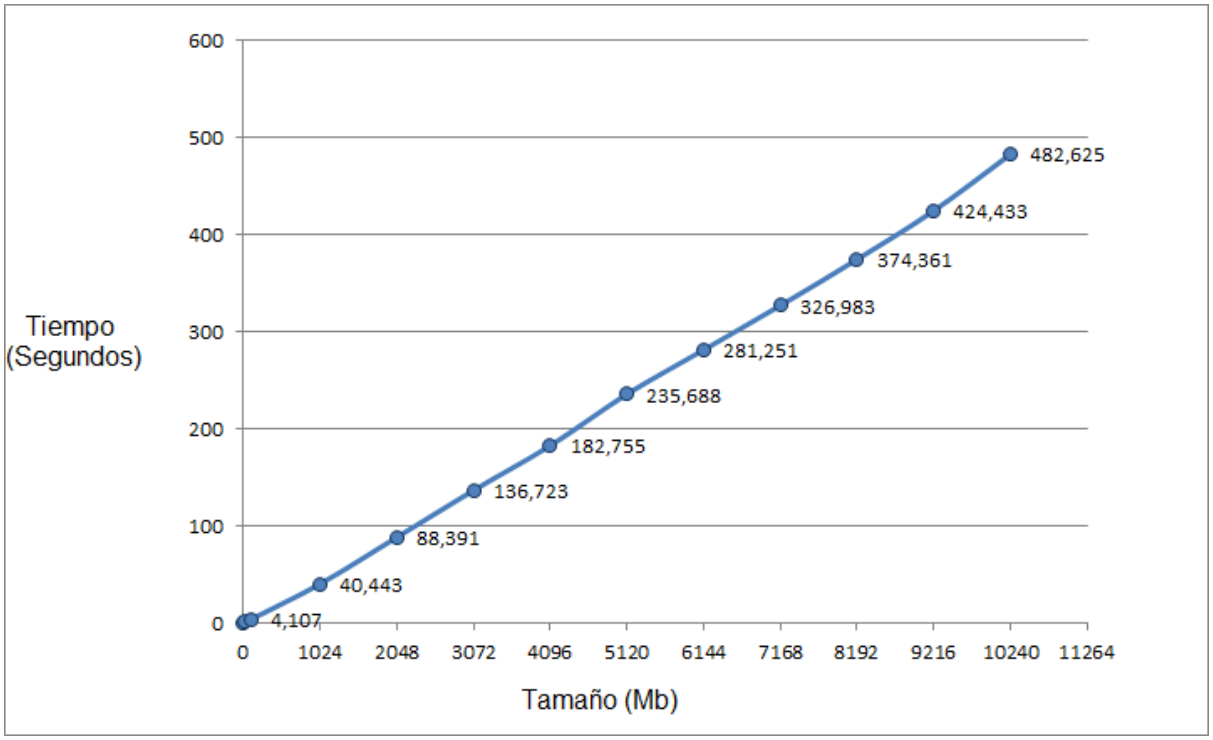

Figura 5-5: Resultados obtenidos de las pruebas realizadas a la capa 3 (hasta 10 GB).

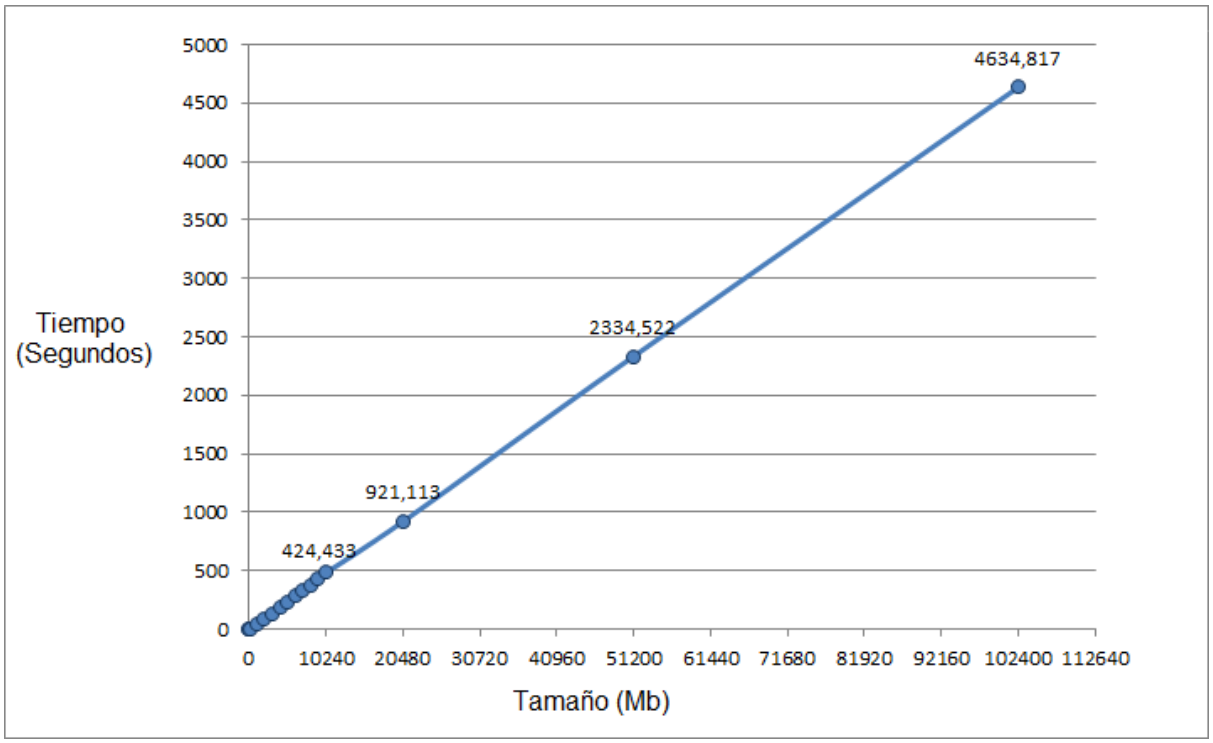

Figura 5-6: Resultados obtenidos de las pruebas realizadas a la capa 3 (hasta 100 GB).

5.2.4. Análisis Comparativo de Resultados

La figura 5-7 presenta la comparación gráfica en entre las tres capas para las pruebas de inserción desde 1MB hasta 10GB. Por otro lado la figura 5-8 muestra las pruebas realizadas en cada capa desde 1MB hasta 100GB. En los gráficos presentados en las Figuras 5-7 y 5-8 se puede ver la comparación de cada una de las implementaciones, MongoDB representado por el color rojo, Mongoose por el verde y Express en azul. Puede notarse a simple vista que la implementación con los mayores tiempos es la capa 3, mientras que las primeras dos capas presentan tiempos muy similares, con algunos mejores para algunos casos y peores para otros.

La principal razón por la cual la segunda capa no solo no es más lenta que la primera sino que obtiene tiempos más cortos en la mayoría de los casos se debe a varias razones relacionadas con la implementación de los scripts de ambas capas y la naturaleza de las mismas. Inicialmente es lógico pensar que Mongoose, al estar construido sobre MongoDB y ejecutarse mediante sus propios controladores, tardará más en ejecutar cada prueba. Esto claramente no se cumple y la razón más probable está dada por el hecho de que se logra ganar tiempo por la forma de ejecución de Node.js. En este sentido, Node.js es de naturaleza asincrónica al igual que la de su subyacente JavaScript, acompañado por el uso de la librería “async” en la ejecución de cada batch o lote de inserciones.

Por otro lado, la tercera capa es claramente más lenta que las anteriores (hasta un 20% según el caso), sus pruebas se ejecutan sobre dos scripts los cuales actúan como cliente y servidor. El tiempo es calculado sumando cuánto tarda cada comunicación más la espera hasta que el servidor responda con la confirmación de la operación exitosa. Como ya se explicó anteriormente, ese retraso se encuentra dado por el tiempo de transferencia de datos mediante HTTP entre los procesos.

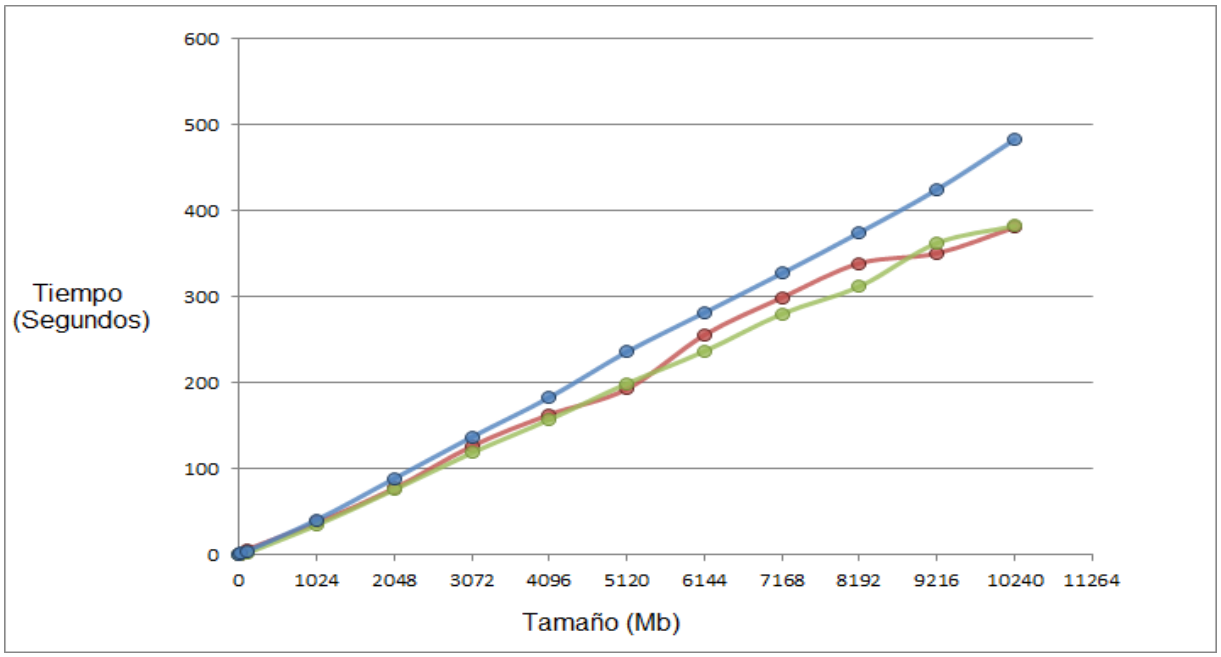

Figura 5-7: Comparativa entre los resultados obtenidos de las pruebas realizadas para cada capa (hasta 10 GB).

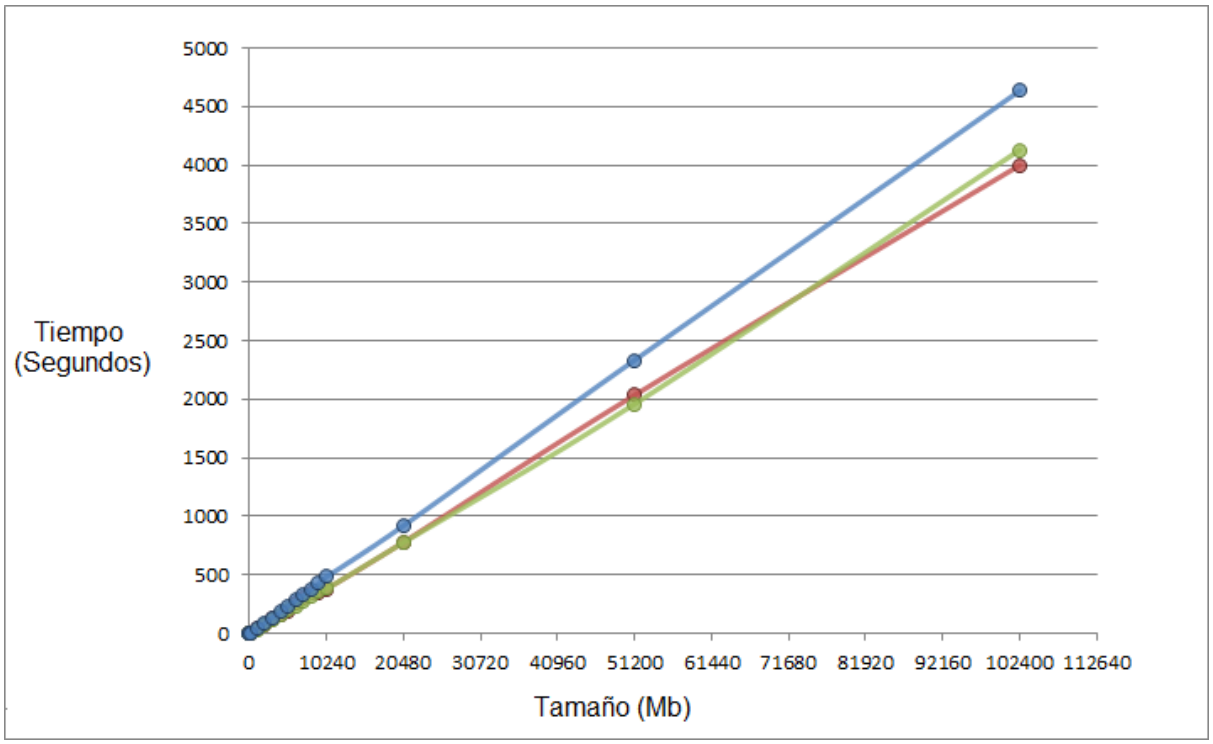

Figura 5-8: Comparativa entre los resultados obtenidos de las pruebas realizadas para cada capa (hasta 100 GB).

Capítulo 6 | Desarrollo y Ejecución de Pruebas por Capa

En el transcurso de este capítulo se introducen los distintos scripts utilizados para las pruebas descritas en el capítulo 5. El objetivo de este capítulo es mostrar el funcionamiento interno de los mismos para explicar cómo se llegó a los resultados obtenidos y entender las diferencias y consideraciones especiales que se tuvieron que tener entre las distintas capas de los experimentos, tanto para su desarrollo como para su ejecución. Para facilitar su lectura, los scripts utilizados se presentan en forma de pseudocódigo.

6.1. Inserción de Datos en la Capa 1: MongoDB

A continuación se muestra el pseudocódigo del script de inserción de datos y medición de tiempo para la primer capa. Esta instancia utiliza la consola de comandos de MongoDB conocida como mongo Shell para su ejecución. La misma es una interfaz interactiva que utiliza JavaScript y permite operaciones sobre los datos y/o administrativas.

```
var timestampTotal = new Date();
var insertionLoop = totalAInsertar / cantidadDeColecciones
var bufferSize= 16; //cantidad de elementos a insertar en un mismo momento
var tiempo = 0;

//Se genera un string aleatorio de 128Kb
var significative_value = randomString();

//Se crean los bulks/lotes para cada colección
var bulkAsig = inicializarBulk();
var bulkTitle = inicializarBulk();
var bulkEmp = inicializarBulk();
var bulkProj = inicializarBulk();

FOR (index = 0 to insertionLoop) {
   //Se crean los JSON de un elemento para cada colección (los documentos a insertar)
   var project = {
      _id: index,
      number: numeric_value,
      name: significative_value,
      budget: numeric_value
   };
   var title = {
      ...
   };
   var employee = {
     ...
   };
   var asignation = {
      ...
   };
```

```
    //Se agregan los JSON en su respectivo bulk/lote de datos
    bulkAsig.insertar(asignation);
    ...
    //Si el bulk/lote se llenó (alcanzó el tamaño de buffer) se insertan los elementos
    if ((i +1 ) % bufferSize== 0) {
          //sumar tiempo de ejecución de cada bulk particular al tiempo total
          var timestampParticular = new Date();
          bulkAsig.ejecutar();
          ...
          tiempo+=(new Date() - timestampParticular );
          //imprimir en pantalla la cantidad insertada en cada bucle
          print('Insertados ' + (index+1) + ' documentos x 4 = ' + (index+1)*4);
    }
}
ENDFOR
//imprimir resultados finales
print('----------FIN----------');
print('Insertados ' + index + ' documentos x 4 = ' + index*4 + ' en ' + (new Date() -
timestampTotal )+ 's');
print('Tiempo de insercion puro: ' + (tiempo + 's'));
```

6.1.1. Funcionamiento Básico del Algoritmo

El algoritmo recibe como único parámetro la cantidad en Megabytes de datos que deben ser insertados. Luego calcula la cantidad de elementos que se deben generar con el fin de igualar dicho tamaño por medio del tamaño de datos definido, en este caso 128Kb. Finalmente genera los registros a insertar (JSON), los cuales son idénticos entre sí a excepción de su identificador único, y almacena las instrucciones de inserción en bulks (lotes). Cuando la cantidad de instrucciones almacenadas en cada bulk alcanza el tamaño de buffer definido, en este caso 16, se ejecuta el bulk insertando efectivamente los elementos y se repite el proceso.
La medición del tiempo de inserción se realiza calculando el tiempo entre el comienzo y fin de cada instrucción de ejecución de los bulk por medio de variables de tipo Date. Las mismas son utilizadas como timestamps y por su diferencia se obtiene el tiempo de ejecución de cada bulk. Finalmente se suma el total de las diferencias para obtener el tiempo de inserción total. Se mide además el tiempo total del algoritmo para observar la diferencia entre el tiempo de inserción puro y el tiempo de ejecución de todo el algoritmo.

6.1.2. Consideraciones

El algoritmo presentado fue diseñado considerando el esquema de datos definido en el capítulo 3 [ver figura 3-1]. Teniendo esto en cuenta, fue necesario dividir el total de datos a insertar por la cantidad de objetos del esquema los cuales luego se traducirán en las colecciones de la base de datos. También se estableció que el tamaño de datos sería de 128Kb, es decir que ese será el peso de cada documento a insertar. Para lograr ese tamaño de datos específico se utilizaron strings, siendo que los mismos son simples de crear con el tamaño deseado. Conociendo que BSON, el formato utilizado para almacenar documentos

de MongoDB, utiliza UTF-8 como el codificador de caracteres para su tipo de dato string [MongoDB Inc, 2017g] y sabiendo que los primeros 128 caracteres de UTF-8 son almacenados en un solo byte [Unicode, Inc, 2015] , se puede calcular la cantidad de caracteres necesarios para que el string ocupe el tamaño deseado. Para el caso de 128Kb son necesarios 131072 caracteres (siempre que su código de caracter se encuentre entre 0 y 127) siendo que los mismos equivalen a 131072 bytes, por lo cual 131072 bytes / 1024 = 128Kb.

6.1.3. Ejecución

Se analizará a continuación un ejemplo de la ejecución del script descrito anteriormente, y como se observa en la Figura 6-1 el script es ejecutado mediante el comando:

- mongo test --eval “var arg1=100” test.js

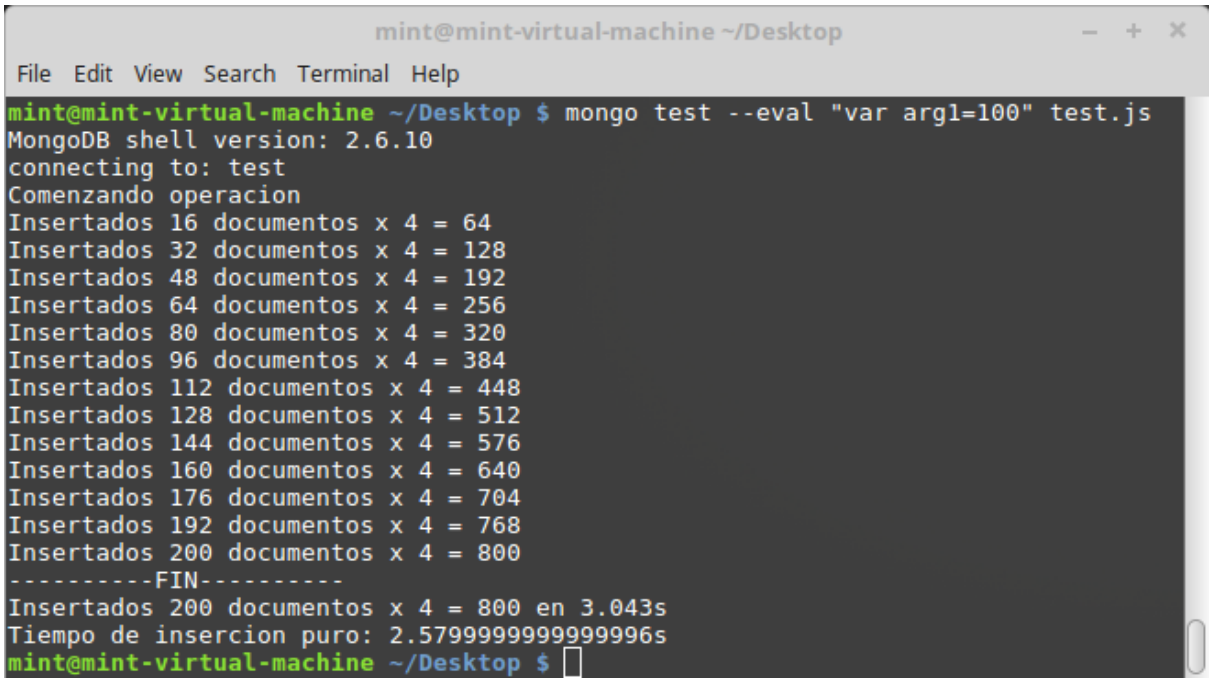

Figura 6-1: ejemplo de ejecución de script para la capa 1 (MongoDB).

Analizando cada componente del comando se obtiene:

1. mongo: El comando mongo inicializa mongo shell y lo conecta con la instancia de MongoDB corriendo en localhost mediante el puerto por defecto. En este caso no es necesario especificar el directorio completo ya que el mismo fue añadido a la variable PATH del sistema durante la instalación de MongoDB [MongoDB Inc, 2017d].
2. test: test es un parámetro que se le pasa al comando mongo y representa el nombre de la base de datos a utilizar.
3. --eval: El parámetro --eval le indica a mongo que debe ejecutar un archivo o expresión JavaScript.
4. “var arg1=100”: Representa la primer expresión a ejecutar, en este caso se utiliza como parámetro para el script inicializando la variable arg1. De esta forma se puede cambiar el valor de arg1 desde afuera del script principal.
5. test.js: Representa el nombre del script a ejecutar.

En el ejemplo se muestra la inserción de 100 MB. Como ayuda visual para ver el progreso de la ejecución, se imprime en pantalla la cantidad de documentos insertados cada vez que se ejecutan las operaciones de un bulk (dependiente del tamaño de buffer). Al haber 4 bulks (uno por colección) y siendo que insertan sus datos en el mismo momento, se muestra por un lado la cantidad de documentos insertados individualmente y por el otro el total. Al finalizar se muestra el tiempo total que tardó la ejecución del script junto con el tiempo que tomó solo la inserción de elementos para apreciar la diferencia entre ambos.

6.2. Inserción de Datos en la Capa 2: Mongoose

A continuación se muestra el pseudocódigo del script de inserción de datos y medición de tiempo para la segunda capa. A diferencia del script para la primera capa, la cual utiliza mongo shell para su ejecución, esta instancia se ejecuta sobre Node.js. Este último es un ambiente de ejecución de JavaScript del lado del servidor basado en eventos y de entrada y salida asincrónica.

```
var start = new Date();
//Inicializar mongoose
mongoose.conectar();
//Generar un string aleatorio de 128Kb
var value = randomString();
var documentNumber = totalAInsertar
//Crear una cola para cada coleccion
var queue1 = [];
...
var bufferSize = 16;

//Encolar todos los elementos en su respectiva cola
function encolarTodo(){
        FOR (var i = 0 to documentNumber) {
                var assignation = new Assignation();
                asignation._id = i;
                asignation.responsability = value;
                asignation.project_id = currentId;
                asignation.employee_id = currentId;
                queue1.push(asignation);

                var proj = new Project();
                ...
                queue2.push(proj);

                var title = new Title();
                ...
                queue3.push(title);

                var employee = new Employee();
                ...
                queue4.push(employee);
        }  ENDFOR
```

```
}

//Dividir las colas en lotes del tamaño de buffer (16) e insertar los elementos
function dividirInsertar(queue1, queue2, queue3, queue4){
        WHILE queue1.length > 0 {
            var lote = queue1.truncar(bufferSize)
            Assignation.insertar(lote);
        }
        ENDWHILE
   print( "-----FIN Asignación----- en "+(new Date() - start+' segundos')
        WHILE queue2.length > 0 {
        ...
        }
        WHILE queue3.length > 0 {
        ...
        }
        WHILE queue4.length > 0 {
        ...
        }
}
//Ejecutar las funciones
encolarTodo();
dividirInsertar(queue1,queue2,queue3,queue4);
```

6.2.1. Funcionamiento Básico del Algoritmo

Al igual que en el script de la primera capa, recibe como único parámetro la cantidad en Megabytes que deben ser insertados para luego calcular la cantidad de elementos que se deben generar con el fin de igualar dicho tamaño según el tamaño de datos establecido (128Kb). A continuación, son generados y almacenados todos los registros a insertar en cuatro colas (una para cada colección). Una vez encolados todos los elementos se divide cada una de las colas según el tamaño indicado por el tamaño de buffer, en este caso 16. Cada trozos de cola es luego insertado en la base de datos mediante la instrucción model.create() propia de Mongoose hasta que la cola se vacíe.
A diferencia de la primera capa, en donde los bulks de todas las colecciones podían ser ejecutados en el mismo bucle, en este script la inserción de cada colección se encuentra separada para cada una de las 4 colas. Esto permite hacer uso del asincronismo de Node.js para independizar la inserción sobre cada colección, lo cual lo convierte en un proceso concurrente. El hecho de que la ejecución sobre cada colección sea independiente implica que el fin de la inserción será distinto para cada una de ellas. Debido a esto, para medir el tiempo de ejecución total, el algoritmo mide el tiempo de inserción de cada colección y se toma como tiempo final el de la última en terminar de ejecutarse.

6.2.2. Consideraciones

Al no trabajar directamente sobre mongoDB, como era el caso para el primer script, se debe establecer una conexión con la base de datos como primer paso de la ejecución. Cabe destacar que en la configuración de conexión se especificó el campo “socketTimeoutMS”

con el valor 0. Esto evita que las inserciones fallen por timeout, efecto agravado por las condiciones de carrera que trae acompañada la asincronía de Node.js.
El algoritmo funciona de manera similar al del primer script con la diferencia de que, al ser que se ejecuta sobre Node.js, fue necesario estructurarlo de forma tal que funcione correctamente de manera asincrónica. Para dicho fin el lenguaje provee herramientas en la forma de promesas y callbacks, pero se utilizó además la libreria async [McMahon, 2017] que facilita la gestión de la sincronía. Otra diferencia es el uso de las instrucciones y estructuras de datos propias de Mongoose en lugar de utilizar las de MongoDB de forma directa. Esto implica a su vez la necesidad de definir los modelos de datos que Mongoose utiliza para realizar las validaciones sobre los mismos, brindando así sus funciones como ODM. A continuación se muestra un ejemplo de definición de modelo para el objeto de tipo empleado del modelo utilizado:

```
// importar modulos
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var ObjectId = mongoose.Schema.Types.ObjectId;

// definir esquema
var EmployeeSchema = new Schema({
    _id: ObjectId,
    name: String,
    title_id: [{type: Schema.ObjectId, ref: "Titulo"}]
});

module.exports = mongoose.model('Employee', EmployeeSchema);
```

6.2.3. Ejecución

Se analizará a continuación un ejemplo de la ejecución del script descrito anteriormente. Como se observa en la figura 6-2 el script es ejecutado mediante el comando:

- NODE_ENV=development nodemon ./bin/ScriptC2.js 24

Al analizar cada componente del comando se obtiene:

1. NODE_ENV=development: es una variable de configuración de node.js que especifica el ambiente en el cual la aplicación será ejecutado, en este caso seteada en “development”. El ambiente “development” o de desarrollo provee algunos beneficios de debugging y por eso es utilizado aquí.
2. nodemon: Nodemon es una utilidad que monitorea cambios en el código fuente y reinicia automaticamente el servidor para mayor comodidad durante el desarrollo del script [Nodemon.io, 2018].
3. ./bin/ScriptC2.js: especifica el nombre del script a ser ejecutado, en este caso el script de nombre ScriptC2 dentro del directorio bin del proyecto.
4. numero: finalmente el script recibe como parámetro un número que representa, al igual que en el script de capa 1, la cantidad de datos a insertar en el orden de Megabytes.

En el ejemplo se muestra la inserción de 24 MB. De manera similar al script anterior, a medida que se ejecutan las operaciones de inserción se notifica en pantalla la cantidad de documentos insertados con éxito. A diferencia del script para la primer capa, en este la inserción de cada colección es independiente, por lo cual se especifica entre paréntesis en

cada caso. Al finalizar se muestra el tiempo en segundos. que tardó la inserción para cada colección. Se utiliza el tiempo de mayor valor para determinar el tiempo total de la prueba.

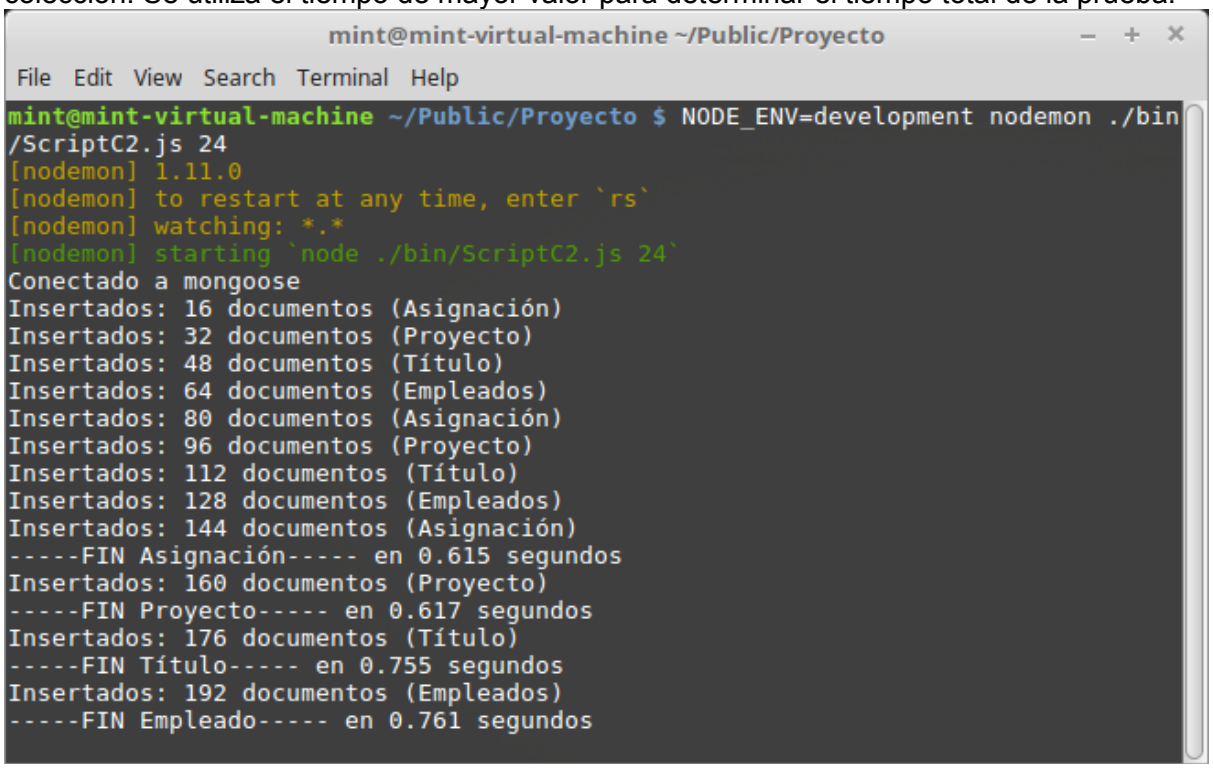

Figura 6-2: ejemplo de ejecución de script para la capa 2 (Mongoose).

6.3. Inserción de datos en la capa 3: Express

A diferencia del script para la primer capa y al igual que la segunda esta instancia se ejecuta sobre Node.js. La diferencia más importante está dada por la importancia de emular la utilización de Express de forma correcta, el cual es el componente que se encarga de facilitar el uso HTTP para atender pedidos y gestionar rutas para los mismos. Para dicho fin el código para las pruebas de la tercer capa se divide en dos scripts principales: uno que actúa de cliente realizando pedidos de inserción y otro de servidor que los atiende. A continuación se muestra el pseudocódigo de los scripts mencionados anteriormente.

Servidor

```
//Cargar paquetes
var express = require('express');
var mongoose = require('mongoose');
var config = require('../config');

//Crear la app de Express y configurar
var app = express();
...

//Definir rutas cada colección
```

```
var assig = require('../routes/assignations');
...

//Conectarse a la base de datos con Mongoose
mongoose.conectar(localhost);

// Registrar las rutas de cada colección sobre /api
app.use('/api/assignations', assig);
...

// Iniciar el servidor
var port = 8000;
app.listen(port);
exports.app = app;
print('Express activo en puerto: ' + port);
```

Cliente

```
var documentNumber = TotalAInsertar / cantidadColecciones;
//Generar un string aleatorio de 128Kb
var value = randomString();
//Definir una cola por colección
var lote1 = [];
...
var bufferSize = 16;
var start = new Date();
//Generar los elementos de cada cola e insertarlos
   FOR ( i = 0 to documentNumber) {
      var assign = {
         '_id' : i,
         "responsibility" : value,
         "project_id" : i,
         "employee_id" : i,
      };
      var proj = {
        ...
      };
      var title = {
        ...
      };
      var employee = {
       ...
      };

      lote1.push(assign);
      ...
//Si se alcanzó el tamaño de buffer se insertan los elementos
      if ( (i+1) % bufferSize == 0 ) {
//Comenzar a medir el tiempo que toma el pedido en ser respondido por el servidor
```

```
        var start = new Date();
//Enviar el pedido con los datos al servidor para cada colección
        request('POST', 'http://localhost:8000/api/assignations/many',lote1);
        …
//Esperar la confirmación exitosa del pedido y sumar el tiempo al contador
        ...
        tiempo+=(new Date() - start);
        print("Insertados: "+i*4+" documentos");
//Vaciar las colecciones para el próximo ciclo
        lote1 = [];
        ...
    }

  }
ENDFOR

print( "-----FIN----- "+(tiempo)+' segundos') ;
```

6.3.1. Funcionamiento Básico del Algoritmo

El script servidor inicializa Express y establece las rutas por las cuales atenderá los pedidos necesarios del cliente. Luego establece la conexión con la base de datos por medio de Mongoose y finalmente define el puerto por el cual escuchará los pedidos, quedando así a la espera de los mismos.
El script cliente se comporta de manera similar al de la primera y segunda capa con algunas diferencias clave. Este recibe el número en Megabytes de la cantidad de datos a insertar. Una vez generados los datos y almacenados en colas del tamaño dictado por el tamaño de buffer, en lugar de ser insertados directamente como en las capas anteriores, se convierte cada cola a un string para finalmente ser enviados al servidor por medio de requests. Cada request es redireccionado por el servidor hacia el enrutador apropiado según la definición de las rutas. Este enrutador traducirá el pedido en el método correspondiente del controlador al cual le corresponde atender el pedido. A continuación se muestran como ejemplos el enrutador y controlador para el tipo de dato asignación.

Router

```
var assigController = require('../controllers/assigController');
var express = require('express');
var router = express.Router();

// '/api/assignations/many'
router.route('/many')
        .post(assigController.createAssigMany);

module.exports = router;
```

Controlador

```
var count=0;
//requerir el modelo
var Assig = require('../models/Asignation');

// Métodos POST para /api/assignations
exports.createAssigMany = function (req, res, next) {

	var rawData = req.body.a.toString().split(";");
	var readyData = [];
	count+=rawData.length;

	while(rawData.length >0){
		readyData.push(new Assig(JSON.parse(rawData.shift())));
	}
	Assig.create(readyData,function(e,result) {
	if(e){console.log(e+"ERROR de asignación"+count);}else{
		console.log("Insertadas: "+count+" asignaciones");}
	});
next();
};
```

Como se puede observar el único método definido del controlador es el de inserción debido a que es el único método necesario para la realización de las pruebas. El método de inserción simplemente vuelve a convertir el cuerpo del pedido de tipo string a los datos originales, validando los mismos mediante Mongoose para ser insertados efectivamente en la base de datos.

6.3.2. Consideraciones

Hizo falta cambiar la configuración por defecto de Express para poder transferir cantidades mayores a 1MB en el cuerpo de cada pedido realizado a través del protocolo HTTP. A pesar de que considerando el tamaño de cada dato y el tamaño de buffer solo se necesitan 2 MB para la transferencia, se estableció un tamaño elevado (100MB) para tener holgura para realizar pruebas con distintas configuraciones. De manera similar al script de la capa 2, al momento de establecer la conexión con la base de datos por medio de Mongoose se lo configura para eliminar el tiempo de timeout para las operaciones.
En el script cliente se optó por incluir la librería “sync-request” la cual permite realizar pedidos HTTP de manera sincrónica, permitiendo así que los mismos se ejecuten de manera secuencial. Este límite que se impone es necesario para evitar consumir toda la memoria con requerimientos, lo cual no solo hace más lenta la ejecución sino que resulta en la eventual terminación del proceso por parte del sistema operativo.

6.3.3. Ejecución

Como se explicó anteriormente, la ejecución se realiza en dos partes: la ejecución del servidor que queda a la escucha de pedidos y la ejecución del cliente que genera los datos y realiza los pedidos de inserción al servidor. Se analizará a continuación un ejemplo de la ejecución de los scripts descritos anteriormente. Como se observa en la figura 6-3 el script es ejecutado mediante el comando:

- NODE_ENV=development nodemon ./bin/serverC3.js

Los componentes del comando, a excepción del nombre propio del script, son los mismos que los utilizados para la ejecución del script para la segunda capa, los cuales ya han sido analizados anteriormente. En el ejemplo se muestra la inserción de 28 MB. Como cada pedido es procesado y ejecutado por su correspondiente controlador, la inserción sobre cada colección es independiente. Por esa razón, a diferencia de los scripts anteriores, se muestran en pantalla los documentos insertados por cada colección junto con el nombre de la misma. Incluso al finalizar la inserción el servidor queda a la espera de pedidos indefinidamente hasta que se cancele el proceso explícitamente. Esto permite realizar las pruebas necesarias desde el programa cliente sin necesidad de resetear volver a inicializar el servidor cada vez.

Ejemplo servidor:

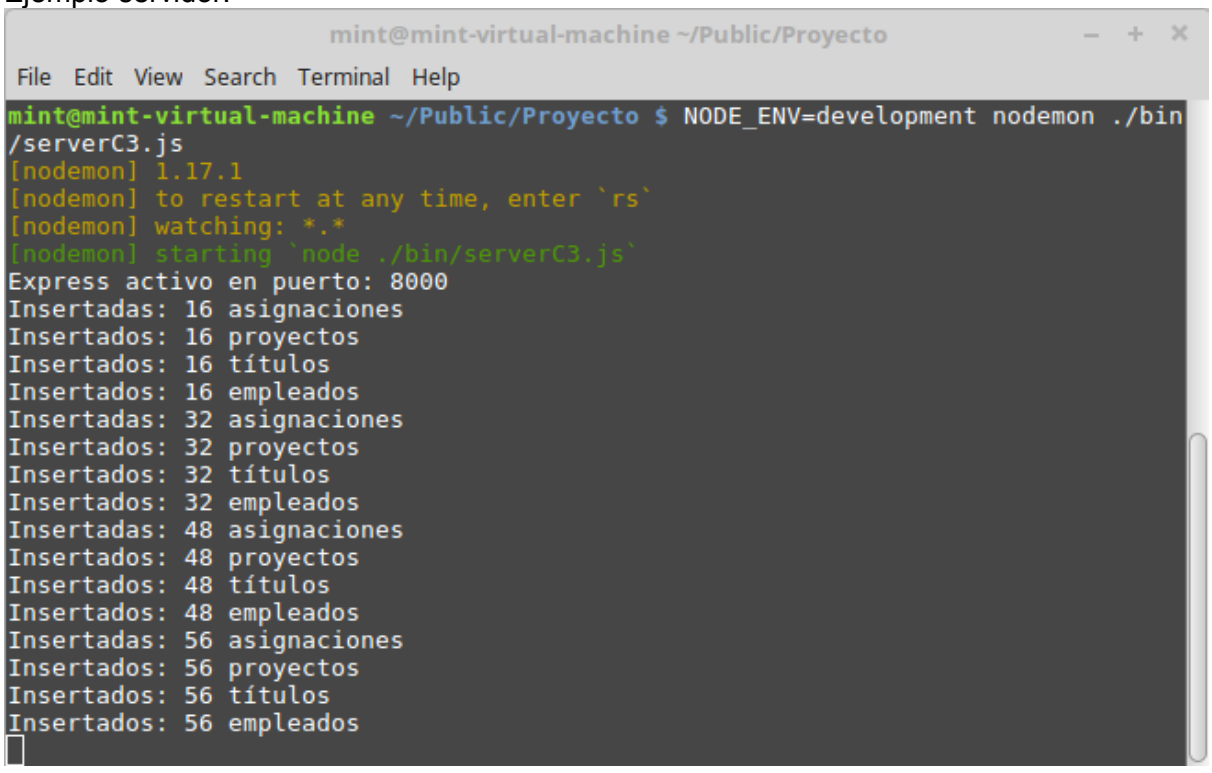

Figura 6-3: ejemplo de ejecución de script servidor para la capa 3 (Express).

Como se observa en la figura 6-4 el script es ejecutado mediante el comando:

- NODE_ENV=development nodemon ./bin/request.js 28

El comando se ejecuta de manera similar que para el script servidor, con la diferencia de que en este se especifica además un número que representa la cantidad en Megabytes a insertar. En el ejemplo se muestra la misma inserción de 28 MB que en el ejemplo anterior pero del lado del cliente. A diferencia del caso anterior solo se imprime en pantalla la

cantidad de documentos insertados total y no se distingue entre la colección particular a la cual pertenecen los datos. Finalmente se muestra el tiempo total que tardó la ejecución del script junto con el tiempo que tomó solo la inserción de elementos para apreciar la diferencia entre ambos.

Ejemplo cliente:

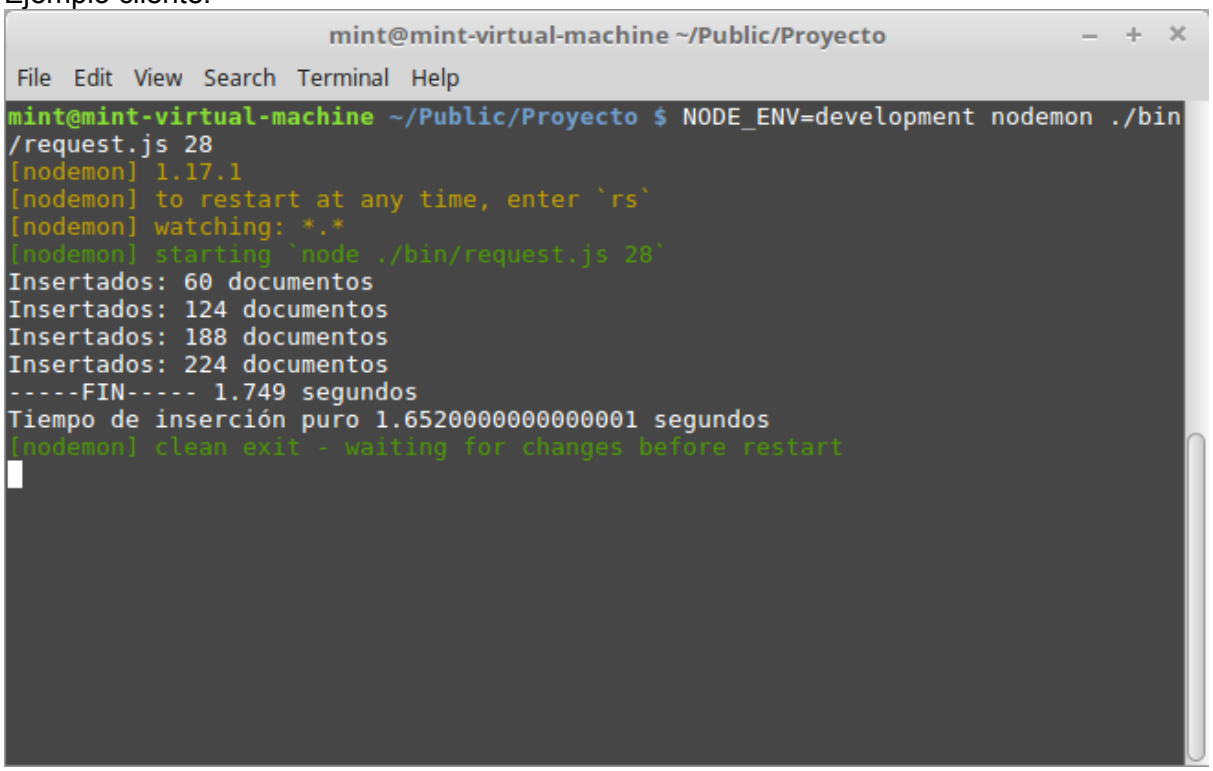

Figura 6-4: ejemplo de ejecución de script cliente para la capa 3 (Express).

6.4. Verificación de Integridad de los Datos Insertados

A continuación se muestra el pseudocódigo del script de verificación de integridad de los datos insertados por cualquiera de los algoritmos tratados anteriormente. Esta instancia utiliza mongo Shell para su ejecución.

```
var documentNumber =TotalARevisar / cantidadColecciones;
var start = new Date();
var index = 0;
var goodCount = 0;
var badCount = 0;
var missing = 0;

//Buscar el primer elemento como referencia
var elementoDeReferencia = db['asignation'].buscarElementoConId(0);

WHILE (index < documentNumber) {
//Comparar los valores de cada campo para cada elemento
```

```
        var asig = db['asignation'].buscarElementoConId(index);
//Verificar si el elemento existe
        if (!asig) {
                missing++;
                print(' MISSING Asignación ' + index);
        }else
//Verificar si el elemento está en buen estado
                if(asig.atributos == elementoDeReferencia.atributos){
                  goodCount ++;
                  print(' Exito Asignación ' + index);
                }else{
                badCount++;
                print(' ERROR Asignación ' + index);
}
//Repetir para cada colección
        var project= db[project].buscarElementoConId(index);
        ...
index++;
}
ENDWHILE
//Imprimir resultados en pantalla
print('Documentos faltantes: ' + missing);
print('Documentos corruptos: ' + badCount );
print('Documentos en buen estado: ' + goodCount );
```

6.4.1. Funcionamiento Básico del Algoritmo

El algoritmo recibe un único parámetro que representa la cantidad de Megabytes de datos que deben ser revisados. Luego calcula la cantidad de elementos a revisar dividiendo el total por el tamaño definido de dato, en este caso 128Kb. A continuación se toma el valor de un elemento para ser utilizado como referencia. Como ya se estableció anteriormente, todos los datos son idénticos entre sí a excepción de su identificador único y su referencia a otros objetos, por consiguiente se pueden verificar todos los datos con el valor de uno solo y el de un índice. Si el elemento existe en la base de datos y los valores de sus campos coinciden con el dato de referencia y su índice correspondiente la verificación se considera exitosa. Si el elemento existe pero el valor de sus campos no coincide con el correspondiente, el dato se considera corrupto y la verificación para ese elemento un fracaso. Si el elemento no existe en la base de datos la verificación se considera un fracaso. Finalmente son impresos en pantalla los resultados obtenidos y el tiempo que tardó la ejecución del script.

6.4.2. Consideraciones

Para que el correcto funcionamiento de este script fuera posible fue necesario estandarizar la asignación de los identificadores únicos de todos los documentos al momento de su inserción. La necesidad de esta restricción sobre los identificadores se debe a que durante la creación de un identificador, si no se especifica un valor, se le asigna uno basado en distintos factores internos del sistema. Para simplificar la iteración sobre los datos y poder predecir rápidamente el siguiente elemento se optó por asignarle manualmente a cada elemento un identificador único basado en su orden de creación. Esto además permite

identificar la posición precisa de un posible elemento corrupto o faltante, facilitando así el debugging de los algoritmos de inserción.

6.4.3. Ejecución

Se analizará a continuación un ejemplo de la ejecución del script descrito anteriormente. Como se observa en la figura 6-5 el script es ejecutado mediante el comando:

- mongo test --eval “var arg1=2” verificador.js

Los componentes del comando de ejecución se corresponden con los de la capa 1 ya que ambos se ejecutan sobre la shell de mongo y se conectan a la misma base de datos [ver ejemplo de ejecución de capa 1]. A diferencia de la primera capa, arg1 ahora cumple un propósito distinto y el nombre del script es otro.

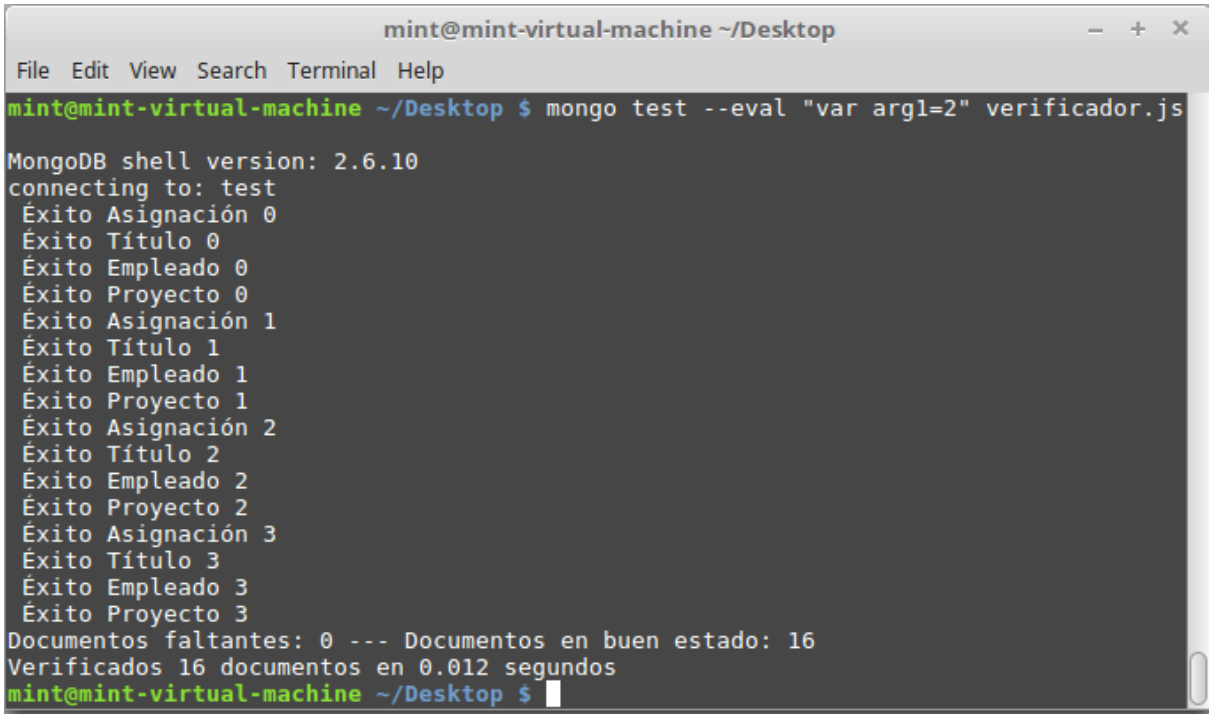

Figura 6-5: ejemplo de ejecución de script de chequeo de integridad de datos.

En el ejemplo se muestra la verificación de 2 MB de datos o 16 documentos (siendo que 16 x 128Kb = 2 MB). Se imprime en pantalla el resultado para cada documento que ya fue revisado junto con su tipo y posición dentro de su colección. En el ejemplo todos los documentos se encontraban en buen estado y por ende se lo indica como “Éxito” pero dependiendo de su resultado pueden verse proyectados como “Corrupción” o “Error” según los criterios explicados en secciones anteriores. Al finalizar se muestra la cantidad de elementos con problemas y la de aquellos en buen estado, junto con el tiempo total que tardó la ejecución del script.

Capítulo 7 | Conclusiones y Trabajos Futuros

A lo largo de este trabajo se ha utilizado la pila de software conocida como “MEAN stack“ para realizar pruebas y analizar el comportamiento y funcionamiento de los distintos componentes que la conforman ante una migración de grandes cantidades de datos. El estudio de este trabajo vio importancia en la utilización de tecnología relativamente nueva y relevante como lo son las bases de datos NoSQL y específicamente MongoDB, siendo el mismo no solo uno de los más populares y en constante desarrollo, sino que también es utilizado por muchas de las compañías más grandes del mundo.
Al abstraer cada componente como una capa que agrega funcionalidad al sistema de software completo, se puede analizar el rol e impacto que tiene cada uno de los mismos en cuanto a los tiempos agregados de cada operación y se obtiene además una idea de dónde se encuentran los principales cuellos de botella. Luego de haberse implementado las soluciones para realizar las pruebas necesarias sobre cada una de estas tres capas y ejecutado dichas pruebas, se pudieron obtener resultados relevantes frente a los objetivos propuestos y distintos a los inicialmente estimados. Además, durante la realización de las pruebas se pudieron identificar los problemas, errores y deficiencias más comunes que presenta la plataforma.

Conclusiones:

Conclusiones durante el desarrollo
En cuanto al desarrollo, se pudo percibir que resulta dificultoso o incluso en muchos casos imposible, la inserción de cantidades o tamaños relativamente (respecto de la cantidad de memoria RAM en particular) grandes de elementos. Para poder hacer este tipo de operaciones, es necesario especificar algún tipo de restricción sobre la inserción (ej: limitar el tamaño de datos a insertar). Esto sucede incluso con estructuras de datos propias de mongoDB como los bulks mencionados en capítulos anteriores, cuya función es la de insertar datos en masa, aunque hacer un uso extensivo de los mismos puede, en algunos casos, saturar la memoria principal.

Conclusiones sobre los resultados de las pruebas
Es lógico suponer que MongoDB, que actúa como la primer capa dentro del sistema de software utilizado, obtendría los resultados de menor tiempo entre las tres capas, con las últimas dos generando un overhead sobre este resultado inicial debido a la funcionalidad que agregan. Sin embargo, como se demostró en el capítulo 5, esto no es del todo correcto al menos a nivel experimental. Mongoose, que representa la segunda capa, fue la que obtuvo los menores tiempos de ejecución en promedio, seguido de MongoDB (capa 1) y en último lugar Express (capa 3).
Los resultados obtenidos están claramente relacionados con que MongoDB corre sobre una plataforma distinta a la de las otras dos capas. Las instrucciones del script que se desarrolló para ella son ejecutadas directamente por mongo shell, que implica una implementación de javascript sincrónica. Mongoose y Express, por otro lado, corren sobre Node.js, cuya implementación de javascript es asincrónica para la entrada y salida, lo cual genera un

ambiente concurrente para la ejecución de las inserciones. Esto no solo provoca un cambio en la ejecución sino que cambia la lógica para el desarrollo de los algoritmos mismos, fenómeno que se pudo ver en detalle en el capítulo 6. Gracias a eso Mongoose obtuvo tiempos 0,2% más rápidos en promedio en comparación con MongoDB a pesar de que esta capa agrega la funcionalidad de validación de datos a partir de los esquemas definidos. No obstante esto, Express cuyos resultados obtuvieron los tiempos de ejecución más largos entre las tres capas como se había previsto.
Las operaciones sobre Express tardan aproximadamente un 16% más de tiempo en ejecutarse en comparación que en la capa 1. Este importante margen está dado por el tiempo de transferencia mísmo del protocolo HTTP, hecho que se comprobó durante la etapa de testeo al probar distintas combinaciones entre la cantidad y el tamaño de los request, donde se obtuvieron resultados casi idénticos.

Trabajos Futuros

Existen líneas de mejora y posibles nuevos desarrollos que pueden ser derivadas a partir de este trabajo. A continuación se plantean algunos de estos posibles trabajos futuros:
1) Estudio comparativo entre MongoDB y una base de datos relacional: Durante el desarrollo de este trabajo y la realización de las pruebas se ha estudiado el comportamiento de MongoDB ante la inserción de grandes cantidades de datos, haciendo énfasis en el rendimiento de los componentes que lo conforman. Se podría poner en marcha un sistema de software basado en una base de datos relacional y realizar un estudio similar al planteado aquí. Con eso se podría realizar una comparativa en detalle entre ambas plataformas en base a todos los aspectos relacionados a la inserción de grandes cantidades de datos. Particularmente estos aspectos irían desde el desarrollo de los algoritmos necesarios, los problemas que conlleva el desarrollo y la realización de las pruebas y sus soluciones, hasta la comparación de los resultados para ambas plataformas.
Este planteo resulta particularmente interesante siendo que las bases de datos NoSQL son a menudo considerados más eficiente en el manejo de Big Data. Una vez realizadas las pruebas necesarias en el sistema relacional se podrían utilizar las herramientas desarrolladas en este trabajo permitiendo comparar ambos modelos sobre una misma máquina. Un ejemplo concreto de lo planteado anteriormente sería establecer un servidor Apache sobre el cual gestionar una base de datos MySQL junto con el uso de PHP como lenguaje de scripting del lado del servidor. Juntando estos tres componentes se obtiene una de las posibles pilas o “stacks” de software de tipo relacional (debido al uso de MySQL) conocidas como LAMP [ver figura 6-1].
Una vez establecidos los componentes a utilizar que formen un stack de software relacional se podría utilizar el mismo esquema propuesto en el capítulo 3 de este documento (ver figura 3-2) para crear la estructura de la base de datos. Habiendo establecido las herramientas a utilizar y estando en condiciones similares entre los sistemas propuestos (relacional y no relacional), será ahora posible la realización de una serie de benchmarks entre ambos sistemas [IBM, 2018b]. Mediante estos distintos sets de pruebas se espera observar las diferencias entre ambos sistemas en distintos aspectos, ya sea velocidad de inserción de elementos, problemas y/o limitaciones de cada uno, etc. Para comparar

directamente entre plataformas enteras se pueden utilizar los resultados para la tercer capa presentes en el capítulo 4 o hacer uso del algoritmo propuesto correspondiente para correr nuevas pruebas para la plataforma no relacional. Para obtener los resultados correspondientes para la plataforma relacional, será necesario la adaptación de los algoritmos planteados en esta tesina al lenguaje correspondiente, en el caso del ejemplo con LAMP este será PHP.

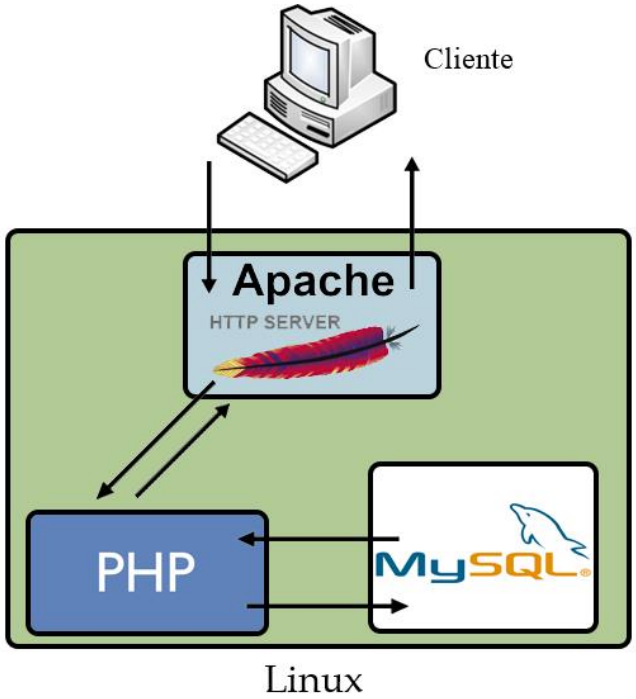

Figura 6-1: Representación gráfica de la pila de software LAMP.

2) Análisis del problema de configuración de MongoDB: Se podría realizar un estudio analizando el problema de memoria de MongoDB tratado en este trabajo, lo cual permitiría tener una idea más clara acerca de la razón por la cual se produce. El estudio podría consistir en una serie de pruebas realizadas sobre una máquina virtual, lo cual permitiría probar de manera sencilla distintas configuraciones de memoria, cambiando la asignación de memoria RAM de la máquina invitada desde el anfitrión. Con esto sería posible medir el punto crítico para cada cantidad de memoria principal que se testea, haciendo uso de los algoritmos desarrollados en esta tesina para probar las distintas combinaciones de tamaño de buffer y de datos y así encontrar la ideal para cada caso. Luego se podrían establecer conclusiones sobre la relación que existe entre cada uno de estos puntos críticos, los cuales podrían representar algún porcentaje particular de la memoria que si se excede produce el error.
También podría darse el caso de que el error solo se produce en sistemas cuyo tamaño de memoria RAM se encuentra por debajo de un tamaño particular. Para llevar a cabo cada prueba en cuestión será necesario modificar alguno de los tres scripts presentados en este trabajo para poder reconfigurar el tamaño de su buffer y/o de elemento. Una vez encontrada la configuración ideal mediante un proceso similar al efectuado en el capítulo 5 de este trabajo, se obtendrá el punto crítico para ese tamaño de memoria principal en particular. Se deberá repetir este proceso para cada tamaño de memoria RAM que se testee. Finalmente

se podrán establecer conclusiones a partir de los resultados hallados para los distintos tamaños de memoria principal que se hayan testeado.

3) Mejora sobre los algoritmos utilizados: Específicamente en el caso del algoritmo de verificación de integridad de los datos insertados, el actual, a pesar de que cumple con su función, no fue el foco principal de este trabajo. Su función es la de verificar que los datos insertados no hayan sufrido modificaciones y asegurando que las pruebas no hayan sido invalidadas debido a algún tipo de corrupción sobre los datos. Es por esta razón que al momento de su implementación, la prioridad no fue la eficiencia en cuanto a su tiempo de ejecución, por lo que se implementó para mongo shell. Sin embargo, como se ha podido observar para las pruebas de la segunda capa, una implementación asincrónica sobre Node.js podría ofrecer resultados de menor cantidad de tiempo. Es por esto que se podría mejorar el algoritmo de verificación de integridad si se lo adaptara para la plataforma Node.js en lugar de su implementación actual sobre mongo shell. Para dicho fin sería necesario adaptar el código, cambiando los usos y asignaciones de objetos por sus accesos correspondientes de Mongoose.

Referencias

1. MongoDB Inc, 2017a. What is MongoDB?. Retrieved from https://www.mongodb.com/what-is-mongodb
2. Mongoosejscom, 2018. Mongoose: Elegant MongoDB object modeling for Node.js, Retrieved from http://mongoosejs.com/
3. StrongLoop, IBM, 2018. Express: Infraestructura web rápida, minimalista y flexible para Node.js. Retrieved from http://expressjs.com/es/
4. Joyent, Inc, 2018. Acerca de Node.js. Retrieved from https://nodejs.org/es/about/
5. Quick Base, Inc, 2018. A Timeline of Database History. Retrieved from http://www.quickbase.com/articles/timeline-of-database-history
6. MongoDB Inc, 2017b. NoSQL Databases Pros And Cons. Retrieved from https://www.mongodb.com/scale/nosql-databases-pros-and-cons
7. MongoDB Inc, 2017c. Flexible enough to fit any industry. Retrieved from https://www.mongodb.com/who-uses-mongodb
8. MongoDB Inc, 2017d. MongoDB Shell. Retrieved 15 April, 2017, from https://docs.mongodb.com/manual/mongo/
9. MongoDB Inc, 2017e. WiredTiger Storage Engine. Retrieved 15 April, 2017, from https://docs.mongodb.com/manual/reference/program/mongod/#wiredtiger-options
10. Van Canneyt, 2005. Embedded databases [PDF file]. Retrieved 24 April, 2017, from https://www.freepascal.org/~michael/articles/embedded1/embedded1.pdf
11. IBM, 2018a. ACID properties of transactions, Retrieved 23 January, 2018, from https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.3.0/com.ibm.cics.ts.productoverview.doc/concepts/acid.html
12. Sadalage & Fowler, 2013a. Impedance Mismatch. In NoSQL Distilled. (pp. 19-20). USA: Pearson Education, Inc.
13. Red Hat Inc., 2018. Hibernate ORM. Retrieved 23 March, 2018, from http://hibernate.org/orm/
14. Berners-lee, 1996. The World Wide Web: Past, Present and Future. Retrieved 6 May, 2017, from https://www.w3.org/People/Berners-Lee/1996/ppf.html
15. Sullivan, 2015. Consistency, Availability, and Partitioning: The CAP Theorem. In NoSQL for Mere Mortals. USA: Pearson Education, Inc.
16. Vaish, 2013. What NoSQL is and what it is not. In Getting Started with NoSQL. (1st ed.). United Kingdom: Packt Publishing Ltd.

17. Sadalage & Fowler, 2013b. The Emergence of NoSQL. In NoSQL Distilled. (pp. 23-25). U.S: Pearson Education, Inc.
18. Ramanathan & Raja, 2013. Restful Service Architecture. In Service-Driven Approaches to Architecture and Enterprise Integration. (pp. 18-23). U.S. IGI Global.
19. Florez, 2018. REST Countries. Retrieved 23 March, 2018, from https://restcountries.eu/
20. AccuWeather Inc, 2018. AccuWeather Enterprise API Documentation. Retrieved 29 March, 2018, from http://apidev.accuweather.com/developers/
21. Fixer.io, 2018. Fixer API. Retrieved 29 March, 2018, from https://fixer.io/documentation
22. Lunaweb ltd, 2018. CloudConvert API. Retrieved 29 March, 2018, from https://cloudconvert.com/api
23. Twitter, Inc, 2018. Docs. Retrieved 29 March, 2018, from https://developer.twitter.com/en/docs.html
24. Google LLC, 2018. Google Drive REST API Overview. Retrieved 29 March, 2018, from https://developers.google.com/drive/v3/web/about-sdk
25. Techopedia Inc, 2018. Software Stack. Retrieved 29 March, 2018, from https://www.techopedia.com/definition/27268/software-stack
26. Mean.io, 2018. Mongo Express Angular Node. Retrieved 29 March, 2018, from http://mean.io/
27. MongoDB Inc, 2017f. Bulk description. Retrieved from https://docs.mongodb.com/manual/reference/method/Bulk/
28. IBM, 2018b. Benchmark testing methods. Retrieved 29 March, 2018, from https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.5.0/com.ibm.db2.luw.admin.perf.doc/doc/c0005059.html
29. IBM, 2018c. What is a database management system?, Retrieved from https://www.ibm.com/support/knowledgecenter/en/zosbasics/com.ibm.zos.zmiddbmg/zmiddle_46.htm
30. MongoDB Inc, 2017g. BSON Types. Retrieved 6 November, 2017, from https://docs.mongodb.com/manual/reference/bson-types
31. Unicode, Inc, 2015. UTF-8 encoding table and Unicode characters Utf8-chartablede. Retrieved from http://www.utf8-chartable.de/unicode-utf8-table.pl?number=1024
32. McMahon, 2017. Async. Retrieved from https://caolan.github.io/async/docs.html
33. Nodemon.io, 2018. Retrieved 29 March, 2018, from https://nodemon.io/

34. Fielding, R. 2000. REST Architectural Elements. In Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.