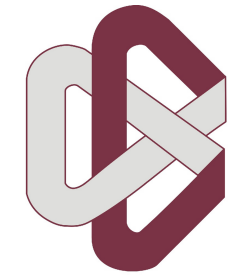
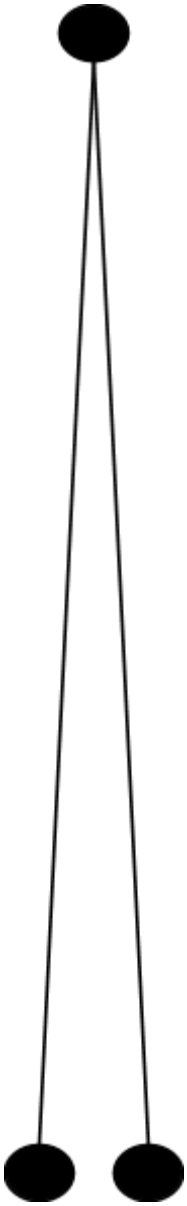


Centro de Investigación en Matemáticas, A.C.

CIMAT



“Modelando datos jerárquicos en bases de datos NoSQL”

Tesis
Para obtener el grado de

Maestro en Ingeniería de Software

Presenta

Agustín Fernando Rumayor Barraza

Asesor

Mtro. José Guadalupe Hernández Reveles

Julio 2014

Resumen

Al tratar de ingresar al movimiento NoSQL, existen diversas formas de modelar la información, lo que lo convierte en un proceso lento y, al elegir entre diversas tecnologías para administrar los datos, puede llegar a ser improductivo sino se tienen claras las características requeridas para el manejo de la información. En esta investigación se implementan seis modelos dentro del ambiente NoSQL (Orientado a documentos, orientado a columnas, clave-valor, grafos, orientado a objetos y multimodelo), y se evalúa el desempeño al abordar un desafío común: el modelado y administración de datos jerárquicos. A través de la comparación se exponen las ventajas y limitaciones de las seis tecnologías seleccionadas y del modelado que representan.

Palabras clave: NoSQL, Orientado a documentos, Orientado a columnas, Grafos, Clave-Valor, Orientado a objetos, Multimodelo, Datos jerárquicos.

Agradecimientos

A mis profesores, familia, compañeros y amigos por compartir sus experiencias, enseñanzas y apoyo.

A los tutores que cada semana durante aportaron consejos y mejoras durante la investigación. Mtro. Pepe Hernández, Mtro. Alejandro García, Dra. Perla Velasco, Dra. Alejandra García , Dr. Arturo Mora, Dr. Sodel Vázquez, Dr. Cuauhtémoc Lemus. Y a toda la comunidad CIMAT por recibirme tan alegremente.

A el CONACyT y COZCyT, por sus programa de becas, que soporta la permanencia de estudiantes e investigadores en la invaluable tarea de generar conocimiento.

Por último agradezco a todos los desarrolladores de software libre por su apoyo a la democratización del conocimiento.

Índice

Índice de Tablas

Índice de Figuras

1 Introducción

Existen diversas aristas en el ambiente del manejo de datos y de las bases de datos, desde el movimiento predominante del modelo relacional, propuesto en los años 70 [1] y popularizado por la utilización de *SQL*[1], y movimientos crecientes como NoSQL y NewSQL que incluyen nuevas formas de organización de los datos y que promueven nuevas formas de crear software [2]. Cada nueva tecnología trae consigo otra manera o una variación al enfrentar los problemas.

Al existir cada día nuevas alternativas agregadas a un sinfín de soluciones, se realizan estudios comparativos, para problemas específicos y generales, que ayudan a la selección de tecnologías para la administración de bases de datos.

Esta investigación está situada en el movimiento NoSQL y propone una comparación exploratoria de las diferentes categorías de las bases de datos más utilizadas: orientadas a documentos, orientadas a columnas, grafos, clave-valor, multimodelo y orientadas a objetos; además de su comportamiento y del desempeño en el manejo de datos jerárquicos.

La investigación se divide en cinco partes. El capítulo 2 expone los trabajos relacionados, las comparaciones realizadas en el ambiente NoSQL y el motivo de esta investigación para extender las comparaciones de los diversos modelos de datos.

En el capítulo 3 se describen las tres principales estrategias para abordar el problema de modelado de jerarquías y árboles, además se puede observar que es un desafío muy estudiado en bases de datos relacionales (listas de adyacencia, path materializado y conjuntos anidados), las implicaciones y ventajas de su uso; algunas de estas estrategias son trasladables a tecnologías NoSQL.

El capítulo 4 describe de forma más detallada las características de las tecnologías seleccionadas para la investigación, desde el año de liberación de la tecnología hasta el propósito de su desarrollo.

Finalmente, en el capítulo 5 se detalla la forma en que fue llevado el experimento de comparación, las implementaciones y mediciones realizadas, y durante el capítulo 6 se analizan los datos obtenidos, se realizan discusiones y se proponen trabajos futuros de la investigación.

[1] Structured Query Language

1.1 Antecedentes

NoSQL es un término ampliamente utilizado para referirse al conjunto de bases de datos no relacionales. Inicialmente el concepto es introducido por Carlo Strozzi al desarrollar un *RDBMS*[2] sin la utilización de SQL como lenguaje de consulta [3]. Años más tarde, el término es reintroducido como lexema para todas las bases de datos que no utilizan el esquema relacional, por lo que han existido propuestas como *Non-Relational* o *NonRel* para una mayor consistencia entre el término y su significado. Actualmente, es el acrónimo mayormente aceptado para "***Not only SQL***" encerrando así, un movimiento de todas las tecnologías emergentes no relacionales. El término NewSQL se refiere a todas aquellas tecnologías nuevas que agregan características y atributos a los sistemas de bases de datos relacionales.

La mayoría de estas tecnologías fueron pensadas como alternativas de escalabilidad, desempeño y disponibilidad de las ya existentes. Manejadores como Cassandra, MongoDB, CouchBase, DynamoDB, Neo4J, OrientDB, entre otras, se hacen llamar así mismas, bases de datos NoSQL. Aún y cuando no se cuenta con una taxonomía oficial [4] para una clasificación homogénea, han existido diversos esfuerzos que proponen una categorización por medio de la estructura de información que guardan. [5][6]

Joe Celko, en su libro "Trees and hierarchies in SQL for smarties" [7], expone de manera amplia los tres modelos generales para representar árboles y jerarquías en bases de datos relacionales. Celko pone en claro las **debilidades y fortalezas** de cada modelo, la complejidad de representar este tipo de estructuras y muestra ejemplos concretos de implementaciones y consultas. Algunos de los modelos descritos son aplicables a algunos modelados NoSQL y otros pueden ser aplicados adaptando las formas de relaciones de los manejadores. Durante el experimento se implementó directamente el modelo de path materializado (Capítulo 3) y dos variaciones del mismo.

En un experimento realizado por Jayathilake, Sooriaarachchi, Gunawardena, Kulasuriya y Dayaratne [8] demuestran la habilidad de cinco tecnologías NoSQL para el manejo de grandes cantidades de información estructurada en árboles con nodos heterogéneos, elegidos aleatoriamente a partir de un conjunto de atributos con diversos tipos de datos, evaluando y representando las operaciones básicas de un árbol: 1) la creación de nodos, 2) la extracción de un sub-árbol, 3) la consulta de los nodos por medio de un atributo y 4) la consulta de los nodos de acuerdo a un valor de un atributo. Este experimento se tomará como base de la investigación, **extendiendo el experimento y las discusiones** al incorporar la base de datos orientada a objetos a los criterios evaluados y **actualizando las bases de datos** utilizadas de acuerdo a su modelado.

[2] Relational Database Management System

Esta y otras comparaciones, ayudan a la toma de decisiones y a la selección de tecnologías, unas enfocadas a las características, que ayudan a la selección cuando se tienen pocas opciones candidatas con propiedades similares, otras comparaciones enfocadas a el grado en que se promueve un atributo de calidad (principalmente desempeño y disponibilidad), y apoyan cuando se busca resaltar un atributo de calidad en la solución. Finalmente, existen comparaciones con un enfoque a observar el comportamiento y exponen un problema específico, lo que ayuda como referencia a soluciones con desafíos similares (Capítulo 2).

1.2 Justificación

Las soluciones conocidas en bases de datos relacionales no resuelven naturalmente con el problema de administración de datos jerárquicos [7]. El conjunto de tablas, con sus columnas y filas, consultan esquemas fijos y guardan información de únicamente entidades y sus atributos. Las relaciones son algo fundamental durante el modelado de jerarquías, y en el enfoque relacional no es una característica principal, basta mencionar que las relaciones son tratadas de la misma forma que un atributo de una entidad, es decir, ambas son una columna dentro de una tabla. La falta de esta distinción, suele hacer más difícil el modelado y diseño de las consultas. Al no encajar directamente el enfoque con el problema, se deriva una mayor complejidad en las decisiones del modelado de los datos, es común encontrarse con dilemas que se contraponen con las características esenciales del modelado en bases de datos relacionales, por ejemplo:

- ¿La jerarquía acepta varios nodos sin padre?
- ¿Es conveniente dejar la columna con valores nulos o mejor agregamos otra tabla?
- Si existen varias relaciones entre la misma tabla, ¿se admitirán *self-joins* para el diseño de consultas o se crean tablas nuevas?
- Si la jerarquía es muy profunda, ¿se denormaliza los datos más consultados?

Generalmente, estas decisiones se inclinan al logro de un desempeño aceptable de la solución. Sin embargo, en las tecnologías clásicas de bases de datos relacionales la consistencia es un objetivo que se encuentra por encima del desempeño deseado y de la disponibilidad de los datos.

A pesar de no lograr el objetivo deseado, la primer alternativa de los diseñadores de software y de su información en un conflicto cotidiano, como es el modelado de datos jerárquicos, se inclina a favor de metodologías tradicionales y conocidas. Tal es el caso del modelo relacional que, a pesar de que ofrece buena estabilidad y aceptación, no es una solución óptima para todos los tipos de problemas que existen. [4]

El ignorar las alternativas emergentes, que puedan encajar al problema de una manera más natural, puede provocar complejidad accidental [9] a la solución que se construye, además de la complejidad característica de transportar un problema informático a una solución tecnológica, decisiones tempranas de tecnologías y de diseño impactan durante todo el desarrollo del producto.

En el experimento base [8] las tecnologías comparadas fueron: 1) Orientadas a documentos, 2) Orientadas a columnas, 3) Orientadas a Grafos, 4) Clave-Valor y 5) Multimodelo. En función del experimento se observó lo siguiente:

- Las bases de datos que soportan el esquema de grafos(Neo4j, OrientDB) fueron las más sencillas de implementar y claramente las de mejor desempeño en las consultas de los datos y jerarquías.
- Durante la creación de nodos, Membase (Clave-Valor) mostró la latencia más baja, siendo la más rápida para la inserción de datos y las de grafos, el peor desempeño.
- En promedio, MongoDB (Orientada a documentos) fue la base de datos más equilibrada entre las cuatro operaciones evaluadas.
- Una vez realizado el experimento se concluyó que es deseable la incorporación de bases de datos orientadas a objetos, otro paradigma diferente, también catalogado como NoSQL, y compararlo con los resultados obtenidos.

Además de extender el experimento base con la base de datos orientadas a objetos, es interesante exponer las implementaciones de las tecnologías con una jerarquía específica y realizar una actualización a las bases de datos utilizadas y de las características y evoluciones de cada tecnología.

El modelado de jerarquías es un tema muy importante dentro de los modelados de información, basta con ver que es un tópico muy discutido en foros, sitios de preguntas y respuestas, listas de correo, tutoriales y artículos tanto en sitios oficiales como en los sitios generales de consulta para programadores. [10][11][12]

El realizar comparaciones para la exploración de los diversos modelos de datos representando jerarquías y árboles, se puede contribuir a la selección de las tecnologías para implementar soluciones, cada problema de datos tiene particularidades y, para el tomador de decisiones, observar implementaciones de un problema cotidiano, sirve como punto de referencia para la toma de decisión en la administración de los datos.

Por último, también se observan áreas de oportunidad en las comparaciones existentes, como es, el agregar más tipos de modelados y tecnologías ante un problema específico, esta investigación trata de abarcar una mayor cantidad de aristas en el universo NoSQL, comparando seis modelos diferentes (Capítulo 4), evaluados conforme a una estructura específica como es la jerarquía de la composición geográfica.

1.4 Objetivos

El propósito de la investigación es la comparación de seis tecnologías NoSQL representando datos jerárquicos, para reconocer la capacidad de cada tecnología como alternativas de solución. Se propone lograr este propósito a través de los siguientes objetivos:

1. Mostrar la implementación de cada tecnología NoSQL ante un problema específico de datos jerárquicos, evidenciando la forma en que la solución se ajusta al problema.

2. Replicar el experimento diseñado con una jerarquía específica, para este caso la composición geográfica a diferencia de datos heterogéneos[8], para contrastar las conclusiones del experimento y su réplica.

3. Extender el experimento base, agregando la base de datos orientada a objetos a las comparaciones y conclusiones.

4. Crear las máquinas virtuales de cada una de las tecnologías seleccionadas para su fácil replicación y que sirvan como base para experimentos posteriores.

Para los objetivos que involucran implementación, se realizó un análisis de las alternativas de cada uno de los modelos de bases de datos NoSQL, para conocer el grado de documentación y popularidad de la tecnología. Todo esto sirve como referencia y si es necesario, pensar en alguna otra alternativa más representativa de alguno de los modelos. (Capítulo 4)

2 Comparaciones de bases de datos NoSQL

Después del año 2000, surgieron una variedad importante de alternativas para la administración de bases de datos no relacionales. Algunas de ellas se desarrollaron y utilizaron con fines de investigación desde los años 60. Hoy retoman fuerza y están siendo utilizadas a nivel comercial [2]. La aparición de esta tecnología ha provocado la creación de diferentes estudios y experimentos que comparan su comportamiento en relación de la tecnologías existentes. En este capítulo se revisan los estudios encontrados relativos a comparaciones de bases de datos no relacionales o NoSQL. En la tabla 2.1 se muestran los cuatro diferentes comparativos comúnmente utilizados, de acuerdo a los objetos de comparación: 1) uno a uno, 2) entre los diferentes modelos NoSQL, 3) tomando un mismo modelo y 4) contra modelo relacional) y los enfoques que pueden tener dichas comparativas. El a) enfoque a características, b) enfoque a un problema en específico y c) enfoque a atributo de calidad.

	a) Enfoque a características	b) Enfoque a un problema en específico	c) Enfoque a atributo de calidad
1) Uno a uno	(RethinkDB [14]), (Abramova [15])	(Pokorny [13]) ([29]	(Badrit [19])
2) Entre modelos	(Han [22])	(Hecht [21])	(Jayathilake [8]), (Narde [20])
3) Mismo modelo	(Cronin [23]), (Ortíz [24])	(Pasarin [26])	(Ciglan [27])
4) Contra relacional	(Bartholomew [28])	(Wei-ping [29])	(Vicknair [30]), (Hadjigeorgiou [31])

Tabla 2.1 Comparaciones de tecnologías NoSQL y sus enfoques

2.1 Comparando tecnologías NoSQL

En esta sección se describe primeramente los cuatro tipos de comparaciones. Los estudios tienen como objetivo obtener un rango de tecnologías para ayudar a la toma de decisiones dependiendo del enfoque que se quiera obtener. Enfocado a las características de las tecnologías (algoritmos, lenguajes de consulta, modelo de datos, etc.), aquellas que demuestren el manejo de un escenario específico (manejo de jerarquías, datos clínicos, datos bancarios, etc.) y aquellos que busquen satisfacer un atributo de calidad en específico (desempeño, disponibilidad, seguridad, etc.). Estos enfoques son descritos al final de este capítulo.

2.1.1 Comparación Uno a uno

Las comparaciones uno a uno son las más comunes, tecnologías como MongoDB y Cassandra, ampliamente aceptadas y las más antiguas, se han convertido en marco de referencia para las tecnologías emergentes. Este tipo de análisis se realiza al momento de la liberación de alguna tecnología comparando su capacidad y características. Existen algunas comparaciones que demuestran un problema específico y los puntos de vista de los dos diferentes modelados [13], en este caso se expone la manera de guardar información de un usuario con Cassandra y BigTable. La comparación realizada por RethinkDB [14] es un ejemplo de cómo una herramienta utiliza otras tecnologías existentes como puntos de referencia. A pesar de encontrar varias comparaciones realizadas por las empresas desarrolladoras, también se pueden encontrar análisis independientes [15], que es un ejemplo de comparación uno a uno entre las dos tecnologías NoSQL más populares [16] (MongoDB y Cassandra), basado en las características, consultas y manejo de transacciones.

Para realizar este tipo de comparaciones en los últimos años se han creado herramientas que ayudan a mostrar las características principales. db-engines.com, vschart.com [17] y findthebest.com [18] son herramientas que presentan, desde la descripción de la tecnología, hasta características específicas que ayudan a tomar decisiones. Cabe mencionar que db-engines, creado y mantenido por Solid-IT, proporciona un ranking basado en la información de la tecnología, sus búsquedas en los motores más populares, el número de ofertas de trabajo, los sistemas que describen trabajar con esa tecnología y el número de profesionales que trabajan con ella. Aunque la mayoría de este tipo de comparaciones tienden a mostrar las características, existen comparativas de atributos de calidad [19] y de cómo reaccionan las tecnologías al respecto.

Este tipo de comparaciones son útiles para tomar decisiones una vez que se conozca el tipo de información que se va a guardar, se haya realizado un filtro previo, teniendo seleccionadas alrededor de tres tecnologías aspirantes para enfrentar el problema.

2.1.2 Comparación Diferentes modelos de datos

Diversas investigaciones estudian de manera exploratoria los modelos NoSQL. El estudio realizado por Narde, Rohan [20] observa el comportamiento de MongoDB (Documentos), Cassandra (Columnas) y DynamoDB (Clave-Valor), en ambientes de nube, realizando las implementaciones en Amazon AWS y Rackspace y midiendo los tiempos de inserción, lectura así como el procesamiento entre 1000 y 500000 registros. La investigación [21] es una de las primeras en que se realiza una exploración de tecnologías a partir de un escenario específico, y presentan tablas comparativas de acuerdo a las características de al menos tres tecnologías por cada modelo (Clave-valor, Documentos, Columnas y Grafos) tanto en las consultas como en el manejo de concurrencia, particiones y replicación. En la

comparación NoSQL [22] se describen las características básicas y se categorizan más de diez tecnologías de acuerdo al teorema CAP[3](Apéndice 1) y expone los antecedentes y limitaciones que afrontan estas tecnologías. El estudio base de la investigación [8] compara cinco diferentes modelos afrontando un escenario específico (manejo de árboles) y medir el desempeño el manejo de nodos.

En promedio, los estudios exploratorios exponen tres aristas del movimiento NoSQL (Clave-Valor, Documentos y Columnas) y existen pocos estudios con un máximo de cinco modelos[8], por lo que existe una oportunidad de extender al número de modelos a evaluar. El principal objetivo de este tipo de análisis, corresponde a la búsqueda de nuevas alternativas para el manejo de la información, principalmente para aquellos que desean voltear al mundo NoSQL. La presente investigación cae dentro de esta clasificación al evaluar seis modelos diferentes.

2.1.3 Mismo modelo de datos

Cuando se tiene claro el modelo de datos que se puede ajustar a los requerimientos, las comparaciones de las tecnologías dentro de una misma categoría dan una perspectiva más profunda y detallada de la decisión. Las categorías NoSQL con más disponibilidad de análisis son las orientadas a documentos, clave-valor y existe un gran interés últimamente por la comparación de las opciones en bases de datos orientadas a grafos.

Devlin Cronin [23] realiza una comparación cualitativa de seis sistemas clave-valor, Berkeley DB, Dynamo, Hyperdex, SILT, LevelDB y Voldemort, las primeras cuatro seleccionadas por su base en investigación científica y las últimas dos por su uso en grandes corporaciones.

Para la comparación de las bases de datos orientadas a documentos, se va a encontrar MongoDB en la mayoría de los estudios, aunque existen más de 15 tecnologías conocidas y categorizadas en diversas listas, merecen ser comparadas más allá del análisis uno a uno con MongoDB. Un ejemplo donde se comparan MongoDB, CouchDB y Terracota es la investigación “Mirada a bases de datos NoSQL de código abierto orientadas a documentos” donde se analizan las formas de almacenaje, replicación, consultas y fragmentación [24].

En términos científicos las bases de datos orientadas a grafos son las más analizadas, existe un gran interés por la gran cantidad de algoritmos que pueden ser aplicados a sistemas de grafos. Tinkerpop Blueprints [25] es un conjunto de implementaciones, similar a los propósitos de JDBC[4] y ampliamente utilizada para realizar comparaciones con grafos, Pasarin Perea [26] demuestra de manera amplia la eficiencia de las tecnologías de grafos para afrontar árboles genealógicos y la investigación [27], expone las limitaciones y ventajas, en términos de desempeño de las tecnologías actuales orientadas a grafos.

[3] Consistency, Availability and Partition tolerance (Consistencia, Disponibilidad y tolerancia a particiones)
[4] Java Database Connectivity

2.1.4 Comparación con sistemas relacionales

Los estudios contra bases de datos establecidas como PostgreSQL, MySQL, MS-SQL, Oracle, suelen ser el primer punto de comparación, al principio del movimiento NoSQL, utilizados para mostrar que es una tecnología alternativa que merece ser vista y actualmente para tener una referencia en cuanto a métricas y formas de implementación y modelado.

Al inicio del movimiento, se necesitaron comparaciones de acuerdo a las características conocidas que proveen los manejadores de bases de datos relacionales, y qué de esas características se conservaban en algunos proyectos NoSQL y qué características adicionales se ofrecían[28]. Otros, describen la secuencia y resultados que obtuvieron a partir de un problema real conocido con algún sistema de base de datos relacional. Motivados por el cambio de administración de datos, documentaron el esfuerzo para pasar a una implementación NoSQL [29]. Conforme al crecimiento de la utilización de tecnologías alternativas, se crearon comparativas enfocadas al desempeño [30] y a la escalabilidad [31].

Este tipo de comparación, además de servir como referencia para tecnologías emergentes, también ayuda a la introducción de aquellos que sólo conocen las bases de datos relacionales. Características como transacciones ACID[5] y particiones de datos, suelen mostrar la manera en que un administrador NoSQL maneja los datos y se exponen las ventajas y limitaciones de una tecnología enfrentando un problema.

2.2 Enfoques

2.2.1 Enfocadas a características

El modelo de datos, consulta, replicación y consistencia son los puntos que se consideran frecuentemente en los estudios [14][22][23]. También es deseable considerar otras características [22] como:

- Plataformas soportadas
- Utilización de Mapreduce
- Métodos de respaldo
- Soporte CAP
- Capacidades máximas
- Soporte técnico

[5] Atomicity, Consistency, Isolation and Durability (Atomicidad, Consistencia, Aislamiento y Durabilidad)

- Utilización de SSD[6] y medios de persistencia
- Utilización de caché
- Manejo de fallos y balanceo
- Múltiples centros de datos
- Requerimientos de sistema

2.2.2 Enfocadas a exponer un problema específico

El ver cómo una tecnología se adapta a un ambiente y problema específico suele ser un punto de apoyo y una buena justificación para tomar una decisión. A pesar de que este enfoque suele ser más subjetivo y suele hacerse de forma cualitativa, dan un panorama de a lo que se podría enfrentar. Es interesante ver los problemas expuestos y la forma en que se ha manejado con las diversas tecnologías, un ejemplo interesante es el estudio realizado para la administración de la información de los usuarios [21], también de las bases de datos orientadas a grafos y su naturalidad de guardar datos jerárquicos, se realizaron estudios amplios para el manejo de árboles genealógicos [26]. La comparación del uso de MongoDB y MySQL en la implementación de un sistema de administración de libros [29], es un ejemplo muy concreto y especializado, además de utilizar conceptos que fácilmente se pueden relacionar con el ambiente de bases de datos relacionales.

2.2.3 Enfocadas en un atributo de calidad

En muchas de las ocasiones, la selección de la tecnología a utilizar es llevada a cabo por el grado que satisface o no un atributo de calidad. A pesar de que el movimiento NoSQL promueve en términos generales la disponibilidad, escalabilidad y desempeño, la mayoría de las comparaciones están enfocadas a demostrar el desempeño, tal es el caso de los estudios [8][19][20][27][30], existen tecnologías [38][39] que incorporan instrumentos para ajustar el grado de ciertos atributos y balancear algunos otros que se tienen comprometidos, como la disponibilidad [32]. Además, existen tecnologías que tienen como estandarte algún atributo, en el caso de MongoDB, muestran el alto desempeño y la alta disponibilidad [33]. Si la elección de una base de datos está en función de un atributo específico, este tipo de análisis se vuelve importante.

2.3 Motivaciones de la investigación

Es importante la evaluación de las tecnologías emergentes, esto puede ayudar a consolidar herramientas valiosas para los diferentes problemas dentro del software y filtrar aquellas que necesiten mayor detalles e incorporaciones de características, que las puedan distinguir de las existentes.

[6] Solid-state drive (Disco de estado sólido)

Existen oportunidades dentro de los estudios de comparación descritos, la mayoría de las comparaciones entre modelos, únicamente se limitan a comparar entre tres y cuatro modelos, lo que motiva a incorporar más modelos, en esta investigación se agregan seis modelos, con ello, se tiene una perspectiva más amplia sobre cómo abordar algún problema, para después decidir sobre detalles y características específicas entre diversas tecnologías.

Además, la oportunidad de implementar un desafío común como la representación de datos jerárquicos, es una motivación al ver las desventajas de las diferentes estrategias usualmente utilizadas (Capítulo 3), además la importancia de esta estructura de datos en el mundo real, y el poder observar el comportamiento y evaluar qué nuevas facilidades nos ofrecen las bases de datos NoSQL para la representación de estos datos.

3 Datos jerárquicos en Bases de Datos Relacionales

La representación de datos jerárquicos y grafos en bases de datos relacionales ha sido discutido desde la introducción del SQL. Uno de los mayores esfuerzos por recopilar las alternativas de representación es realizado por Joe Celko [34]. En este capítulo se discuten brevemente los tres principales formas de presentación propuestos por Celko. A continuación, se describe cada forma con un ejemplo práctico en el modelo relacional y se discuten sus implicaciones. Ello da pie y justifica el porqué de la búsqueda de alternativas de modelado en tecnologías NoSQL.

Para explorar las tres formas de representación en el modelo relacional utilizaremos la jerarquía de país, estado, ciudad (Figura 3.1). La elección de esta jerarquía es porque se considera un ejemplo de cómo la información puede crecer y convertirse en un problema de *Big Data*[7], tanto en la flexibilidad de los datos, al incorporar un grado más de localización como códigos postales o colonias, como en el crecimiento exponencial de la cantidad de datos guardados. A pesar de ser un modelo con datos estáticos, en los que cambian ocasionalmente los nodos y sus relaciones a lo largo del tiempo, las consultas y discusiones descritas son tratadas como datos más dinámicos y flexibles.

Las consultas SQL desarrolladas durante el ejemplo fueron validadas y ejecutadas en la herramienta de pruebas SQL Fiddle [35] desarrollada por Jake Feasel (http://sqlfiddle.com/), corriendo MySQL 5.5.32

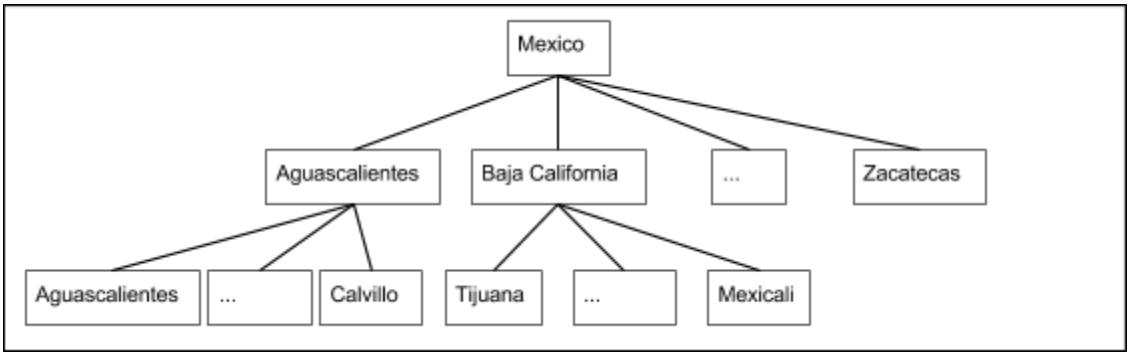


Figura 3.1 Datos utilizados como ejemplo en los modelos de implementación jerárquica.

En los apartados siguientes se explicarán con detalles algunas de las estrategias de implementación y modelado de datos jerárquicos en el ambiente relacional.

[7] Fenómeno derivado de recolectar, buscar, guardar, analizar y comparar grandes cantidades de información.

3.1 Listas de adyacencia

Esta estrategia de implementación está basada en la representación de un grafo por medio de la descripción de los nodos vecinos de un nodo en particular. Durante la implementación en el modelo relacional se realiza describiendo la jerarquía más inmediata hacia arriba. Es el modelo más sencillo de implementar, las únicas decisiones a las que hay que enfrentar durante el modelado de los datos es el grado de normalización, decisión que afecta a las operaciones básicas en una jerarquía explicadas a continuación.

3.1.1 Implementación

```
CREATE TABLE PAIS (pais varchar(50), PRIMARY KEY (pais));
CREATE TABLE ESTADO (estado varchar(50), pais varchar(50), poblacion int, PRIMARY KEY
(estado, pais));
CREATE TABLE CIUDAD (ciudad varchar(50), estado varchar(50), pais varchar(50), PRIMARY
KEY (ciudad, estado, pais));
```

País
México

Estado	País	Población
Aguascalientes	México	1184196
Baja California	México	3155070
Zacatecas	México	1490668

Ciudad	Estado	País
Aguascalientes	Aguascalientes	México
Calvillo	Aguascalientes	México
Tijuana	Baja California	México
Mexicali	Baja California	México

Figura 3.1.1.1 Implementación de la jerarquía con listas de adyacencia

3.1.2 Agregando Nodos

La inserción de datos es una de las mayores ventajas del modelo, conociendo e insertando la clave del nodo padre es lo único que se tiene que realizar. Se pueden agregar limitaciones para asegurar la integridad de la jerarquía.

```
INSERT INTO CIUDAD VALUES ('Jesus Maria', 'Aguascalientes', 'Mexico' );
```

Figura 3.1.2.1 Inserción de nueva ciudad en listas de adyacencia

3.1.3 Borrando Nodos

El borrado simple de un nodo ocasiona que los nodos hijos queden desconectados de la jerarquía, para evitarlo se debe elegir una de las siguientes opciones:

1. Todos los nodos hijos son removidos.
2. Todos los nodos hijos son enviados con el ancestro.
3. Un hijo toma el lugar del padre.

Para cualquiera de las decisiones es necesario el desarrollo de un *trigger*[8] o la eventualización en cascada de los nodos hijos.

```
CREATE TABLE ESTADO (estado varchar(50), pais varchar(50), poblacion int,
PRIMARY KEY (estado, pais), FOREIGN KEY (pais) REFERENCES PAIS (pais) ON DELETE
CASCADE);
```

Figura 3.1.3.1 Evento de borrado en cascada

3.1.4 Actualizando Nodos

Las actualizaciones, así como en el borrado, es necesario la eventualización en cascada (Figura 3.1.4.1) si es que se actualizan los campos de referenciación.

```
CREATE TABLE ESTADO (estado varchar(50), pais varchar(50), poblacion int, PRIMARY KEY
(estado, pais), FOREIGN KEY (pais) REFERENCES PAIS (pais) ON UPDATE CASCADE);
```

Figura 3.1.4.1 Evento de actualización en cascada

3.1.5 Consultando Nodos

Si se encuentran los datos normalizados como en el ejemplo (Figura 3.1.5.1) es sencillo ver la profundidad de la jerarquía al observar la cantidad de tablas de los nodos sino, es necesario conocer un nodo hijo y contar los saltos hacia un nodo raíz. Para la consulta de una porción de la jerarquía se complica, la utilización de ciclos o de recursividad es necesario.

```
SELECT ESTADO.poblacion, CIUDAD.ciudad FROM CIUDAD LEFT JOIN ESTADO ON ESTADO.
estado = CIUDAD.estado;
```

Figura 3.1.5.1 Consulta de jerarquía

La mayoría de la cantidad de consultas que se necesitan en este modelo son anidadas o con un número de *joins*[9] similar a la profundidad de los datos, en datos no normalizados es común el uso de *self-joins*[9] para este modelo. Una de las variantes para mejorar el desempeño en las búsquedas de nodos hijos es cambiar el flujo de la información, aunque es necesario que el manejador que se utilice soporte consultas recursivas para una buena implementación del modelo.

[8] El disparador o trigger es un componente de una base de datos asociado a una tabla y activado en un evento específico.
[9] Sentencia SQL para combinar registros en diferentes tablas o en la misma tabla (self-join)

3.2 *Path* materializado

Este modelo se encarga de concatenar el camino más corto de algún nodo raíz hasta el nodo, para aquellas jerarquías con características de árbol, existe un solo camino posible de la raíz hasta el nodo, existen dos tipos de formas principales con algunas variantes cada una, la numeración de relaciones y la numeración de nodos. El más común y el que vamos a describir es el de nodos.

3.2.1 Implementación

```
CREATE TABLE PAIS (pais varchar(50));
CREATE TABLE ESTADO (estado varchar(50), path varchar(200), poblacion int);
CREATE TABLE CIUDAD (ciudad varchar(50), path varchar(200));
```

País
México

Estado	Path	Población
Aguascalientes	México	1184196
Baja California	México	3155070
Zacatecas	México	1490668

Ciudad	Path
Aguascalientes	México/Aguascalientes
Calvillo	México/Aguascalientes
Tijuana	México/Baja California
Mexicali	México/Baja California

Figura 3.2.1.1 Implementación de path materializado

3.2.2 Agregando Nodos

La inserción dentro del modelo es sencillo, únicamente hay que conocer todo el camino hasta el nodo raíz, puede ser consultado a sus hermanos o padre. En la figura 3.2.2.1 está un ejemplo para agregar una ciudad al estado sin conocer todos los ancestros, únicamente el padre al que se va a agregar el nodo, para este ejemplo se utilizó la función de concatenación que se encuentra en la mayoría de manejadores de bases de datos SQL.

```
INSERT INTO CIUDAD SELECT 'Jesus Maria', CONCAT (path, "/", estado) FROM
ESTADO WHERE estado = 'Aguascalientes';
```

Figura 3.2.2.1 Ejemplo de inserción utilizando path materializado

3.2.3 Borrando Nodos

A pesar de que se siguen utilizando eventos o triggers para el borrado de los nodos, es un poco más sencillo que con las listas de adyacencia. También se tiene que decidir si los hijos se borran, son pasados al ancestro más inmediato o un hijo toma el lugar del padre.

Este es un ejemplo de cómo se vería si se borra un nodo y sus hijos son agregados al ancestro inmediato:

```
-- cambio de los hijos --
UPDATE CIUDAD SET path = REPLACE(path, '/Aguascalientes', '') WHERE path LIKE
'%Aguascalientes';
-- borrando al padre --
DELETE FROM ESTADO WHERE estado = 'Aguascalientes'
```

Figura 3.2.3.1 Borrando jerarquías y agregando al ancestro más próximo

3.2.4 Actualizando Nodos

Es necesario, dependiendo de la estrategia elegida, realizar varias operaciones al momento de mover un nodo de un lugar de la jerarquía a otro.

```
-- Moviendo hijos a otro nodo --
UPDATE CIUDAD SET path = CONCAT((SELECT path FROM ESTADO WHERE estado =
'Zacatecas'), '/', 'Zacatecas') WHERE path LIKE '%Aguascalientes';
```

Figura 3.2.4.1 Ejemplo de traslado de nodos

3.2.5 Consultando Nodos

Varias de las consultas están relacionadas con el manejo de cadenas de caracteres, las consultas se vuelven más amigables a medida que funciones como REPLACE, REPEAT, RPAD, RIGHT, LEFT, REVERSE, MID, entre otras, son aplicadas para consultas.

Consultando la profundidad máxima de la jerarquía:

```
SELECT MAX(LENGTH(path) - LENGTH(REPLACE(path, '/', ''))+1) FROM ciudad;
```

Figura 3.2.5.1 Profundidad de la jerarquía

Obteniendo la descendencia de un nodo:

```
SELECT * FROM CIUDAD WHERE path LIKE '%Aguascalientes%';
```

Figura 3.2.5.2 Descendientes de un nodo

En la implementación del path materializado se vuelve más sencillas las operaciones al únicamente manejar un sólo campo. Sin embargo, hay que tener en cuenta que el administrador de base de datos que se seleccione, contenga los métodos para el manejo de textos, y si es posible la utilización de expresiones regulares.

3.3 *Conjuntos Anidados*

Es un modelo que se enfoca en la identificación de los niveles de la jerarquía. Usualmente se utilizan óvalos cuyo contenido representan los hijos (Figura 3.1.1), para muchos conjuntos de datos es un enfoque más natural y por consecuencia entendible, y más escalable que los modelos descritos con anterioridad, es una alternativa que ha sido fuertemente aceptada. Cada nodo es descrito con los nodos que contiene de izquierda a derecha.

3.3.1 Implementación

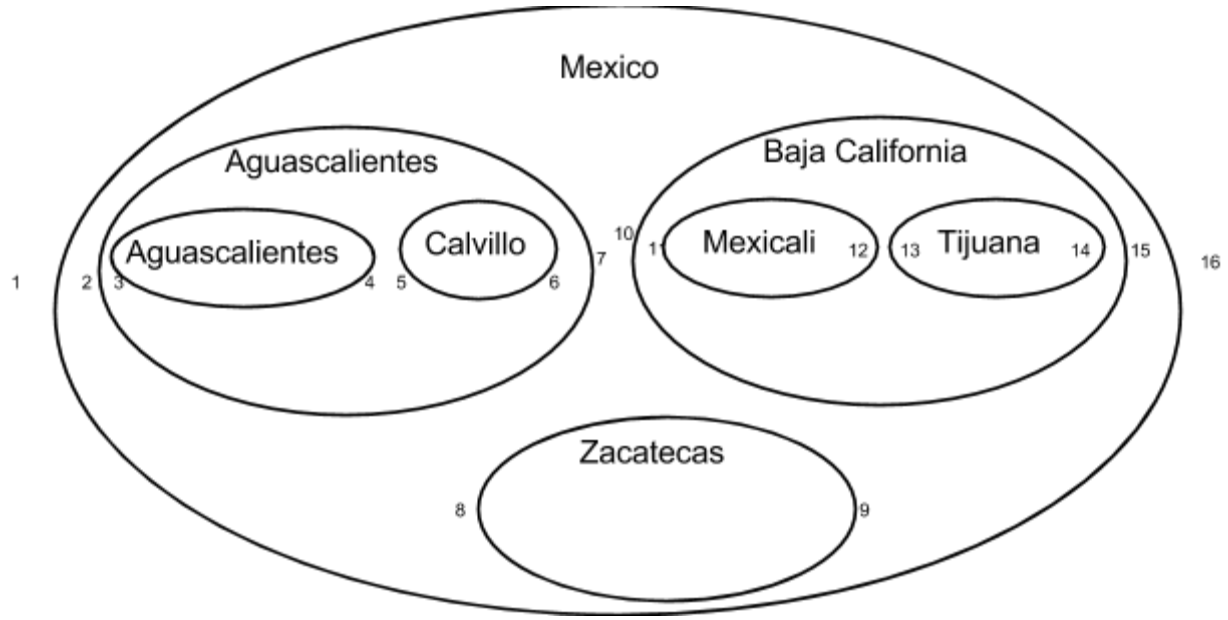


Figura 3.3.1.1 Representación de conjuntos anidados

```
CREATE TABLE PAIS (pais varchar(50), izq int, der int);
CREATE TABLE ESTADO (estado varchar(50), izq int, der int, poblacion int);
CREATE TABLE CIUDAD (ciudad varchar(50), izq int, der int);
```

País	Izq	Der
México	1	16

Estado	Población	Izq	Der
Aguascalientes	1184196	2	7
Baja California	3155070	10	15
Zacatecas	1490668	8	9

Ciudad	Izq	Der
Aguascalientes	3	4
Calvillo	5	6
Tijuana	11	12
Mexicali	13	14

Figura 3.3.1.2 Implementación de la jerarquía con conjuntos anidados

3.3.2 Agregando Nodos

Al momento de necesitar incluir nuevos nodos se necesitan actualizar todos los nodos que se encuentren a la derecha del nuevo nodo, tanto los campos de izquierda y derecha se necesitan incrementar por dos.

```
-- Obteniendo el valor donde se quiere insertar el nuevo nodo --
SELECT @derecha:= der FROM CIUDAD WHERE ciudad='Calvillo';

-- actualizando los nuevos valores de los nodos de la derecha --
UPDATE CIUDAD SET der = der+2 WHERE der > @derecha

UPDATE CIUDAD SET izq = izq+2 WHERE izq > @derecha

INSERT INTO CIUDAD VALUES ('Jesus Maria', @derecha +1, @derecha+2);
```

Figura 3.3.2.1 Ejemplo de inserción de datos usando conjuntos anidados.

3.3.3 Borrando Nodos

Cuando se utiliza el modelo de conjuntos anidados es también común el hecho de aceptar que el remover un nodo de la jerarquía significa también remover los nodos hijos, cualquiera otra decisión significa que de manera costosa se actualicen todos los campos hacia donde se requiera mover los nodos hijos.

```
--Obteniendo los valores del nodo a remover--
SELECT @izquierda:= izq, @derecha:=der, @ancho:=der-izq+1 FROM CIUDAD WHERE
ciudad='Jesus Maria';

-- Removiendo nodo y sus hijos --
DELETE FROM CIUDAD WHERE izq BETWEEN @izquierda AND @derecha;

-- Actualizando los nodos de la derecha
UPDATE CIUDAD SET der = der-@ancho WHERE der > @derecha

UPDATE CIUDAD SET izq = izq-@ancho WHERE izq > @derecha
```

Figura 3.3.3.1 Ejemplo de borrado de un nodo y sus hijos

3.3.4 Actualizando Nodos

El movimiento de la jerarquía se realiza de manera similar al insertado y borrado de los nodos (Figura 3.3.3.1), las implementaciones suelen ser un conjunto de transacciones que involucran las acciones descritas anteriormente.

3.3.5 Consultando Nodos

Teniendo los caminos de izquierda y derecha en cada uno de los nodos, es posible obtener algunas de las funciones particulares de los árboles, como la navegación, al poder utilizar el preorden, y también permite obtener subconjuntos de datos.

Aquí un ejemplo para obtener todos los descendientes de un nodo:

```
-- Obteniendo el nodo --
SELECT @izquierda:=izq, @derecha:=der FROM ESTADO WHERE estado = 'Baja
California';
-- Buscando descendientes --
SELECT * FROM CIUDAD WHERE izq BETWEEN @izquierda AND @derecha;
```

Figura 3.3.5.1 Descendientes en los conjuntos anidados

Los nodos sin descendientes son un problema que en el modelo de listas de adyacencia es complicado y costoso de resolver.

```
SELECT * FROM PAIS WHERE der-izq = 1;
SELECT * FROM ESTADO WHERE der-izq = 1;
SELECT * FROM CIUDAD WHERE der-izq = 1;
```

Figura 3.3.5.2 Consultando nodos hoja

3.4 Comparación de modelos

A partir de la bibliografía consultada [34] y de las implementaciones realizadas, se pueden concluir los siguientes puntos para cada uno de los modelos:

Listas de Adyacencia

- Fortalezas: Implementaciones pequeñas y necesidad de alta escritura y borrado.
- Debilidades: Cuando la jerarquía tiene un número de niveles considerable (mayor a 6) y cuando se requieren consultas de subjerarquías.

Path Materializado

- Fortalezas: Realizar movimientos dentro de la jerarquía, consulta de subjerarquías verticales (hijos, nietos, …).
- Debilidades: Utilizar manejadores que no tengan funciones de manejo de texto y grandes jerarquías.

Conjuntos Anidados

- Fortalezas: Todas las consultas son vistas dentro de la teoría de conjuntos y navegación de la jerarquía sencilla.
- Debilidades: El desempeño de inserciones o actualizaciones disminuye exponencialmente conforme al número de nodos en la jerarquía.

	Listas de Adyacencia	Path materializado	Conjuntos anidados
Agregar nodos			
Borrar nodos			
Mover nodos			
Consultas de nodos			
Consultas de jerarquías			

	Implementación sencilla y buen desempeño
	Implementación sencilla o buen desempeño
	Implementación complicada y mal desempeño

Tabla 3.4.1 Comparación de los modelos con respecto a las operaciones básicas

En la tabla 3.4.1 se muestra un resumen de las ventajas y limitaciones respecto a la implementación y el desempeño formas de presentación al realizar las operaciones básicas. A pesar de que los datos jerárquicos no encajan naturalmente con las bases de datos relacionales el esfuerzo por crear modelos que relacionen el problema con la solución se debe a la estabilidad y madurez de la tecnología y a la amplitud del uso de SQL.

Existen variaciones e híbridos de los tres modelos para conjuntos de datos y problemas más específicos. La creación de eventos y *triggers* para el buen manejo de entrada y salida de los datos y el establecimiento de *restricciones* dentro del manejador de base de datos para conservar las propiedades jerárquicas, son puntos que hay que considerar durante la implementación para conservar la integridad de los datos.

En esta investigación no se incorporan estos modelos a la comparación ya que, se pretende comparar las implementaciones y las nuevas formas de representar datos en bases de datos NoSQL, además son modelos que se conocen ampliamente las ventajas y desventajas de su uso en bases de datos relacionales.

4 Selección de tecnologías NoSQL

Cada día emergen nuevas tecnologías para la administración de datos, algunas con propósitos muy específicos (SciDB, HyperGraphDB) y otras con propósitos generales (MongoDB, Cassandra). Implementaciones más específicas a partir de software libre como Accumulo y Couchbase, proponen alternativas adicionales a las ya existentes. Actualmente hay más de 150 tecnologías NoSQL[5] y varias aristas dentro del ecosistema, a pesar de que todavía no hay una categorización para todos los tipos de manejadores actuales y futuros [4], se pueden delimitar al menos cinco tipos ampliamente aceptados:

- Orientada a columnas
- Orientada a documentos
- Clave-valor
- Orientada a grafos
- Orientada a objetos

Vale la pena mencionar que existe una tendencia reciente de clasificación NoSQL, el modelo híbrido o multimodelo. Importantes exponentes lo respaldan y por lo mismo es importante agregarlo dentro de las comparaciones. Tal es el caso de Couchbase, OrientDB, ArangoDB y FoundationDB. Existen otros modelos con menor popularidad, y que no se agregan a este estudio, la mayoría por tener un carácter más específico o modelados y esquemas fijos como son el multivalor, los multidimensionales y el XML.

En este capítulo se presenta información general de las tecnologías NoSQL a utilizar en el experimento. Se seleccionaron seis manejadores de bases de datos NoSQL de diferente tipo. MongoDB, Cassandra, OrientDB, Neo4J, Membase que corresponden al experimento tomado como base que realiza la comparación del manejo árboles heterogéneos [8]. ZODB es agregada para comparar directamente implementaciones orientadas a objetos y así agregarla a la discusión del manejo de datos jerárquicos. Membase es ahora Couchbase, y ahora incorpora además de clave-valor, el orientado a documentos, por lo que ahora es un multimodelo, para que el experimento aún contenga el modelo de clave-valor, y evaluar los diferentes puntos de modelado, se decidió utilizar Redis, que en el momento de la investigación es la base de datos clave-valor más popular dentro de DB-Engines[36].

	MongoDB	Cassandra*	OrientDB*	Neo4J*	Redis	ZODB
Modelo	Documento	Columnas	Documento y Grafos	Grafos	Clave - Valor	Objetos
Licencia	AGPL	Apache 2.0	Apache 2.0	GPL	BSD	ZPL
Primer liberado	2009	2008	2010	2007	2009	2005
Leng de desarrollo	C++	Java	Java	Java	C	Python
Mejor uso	Multipropósito	Sistema distribuido con mucha operación escritura	Sistemas con altos cambios de esquemas	Sistemas con alta consulta de relaciones	Aplicaciones en tiempo real.	Multipropósito

Tabla 4.1 Bases de datos NoSQL seleccionadas

*Community Edition[10]

En la tabla 4.1 se muestran las tecnologías seleccionadas y sus atributos por las que fueron desarrolladas, así como una breve descripción del mejor uso por el que fueron creadas. Todas las ediciones utilizadas son basadas en Software con licencia Open Source, sin embargo hay algunas con una edición empresarial que agregan mayor capacidad de soporte y algunas características extras para el mantenimiento de los servicios.

[10] Se utilizaron las ediciones de software libre, estas tecnologías también cuentan con ediciones para empresas.

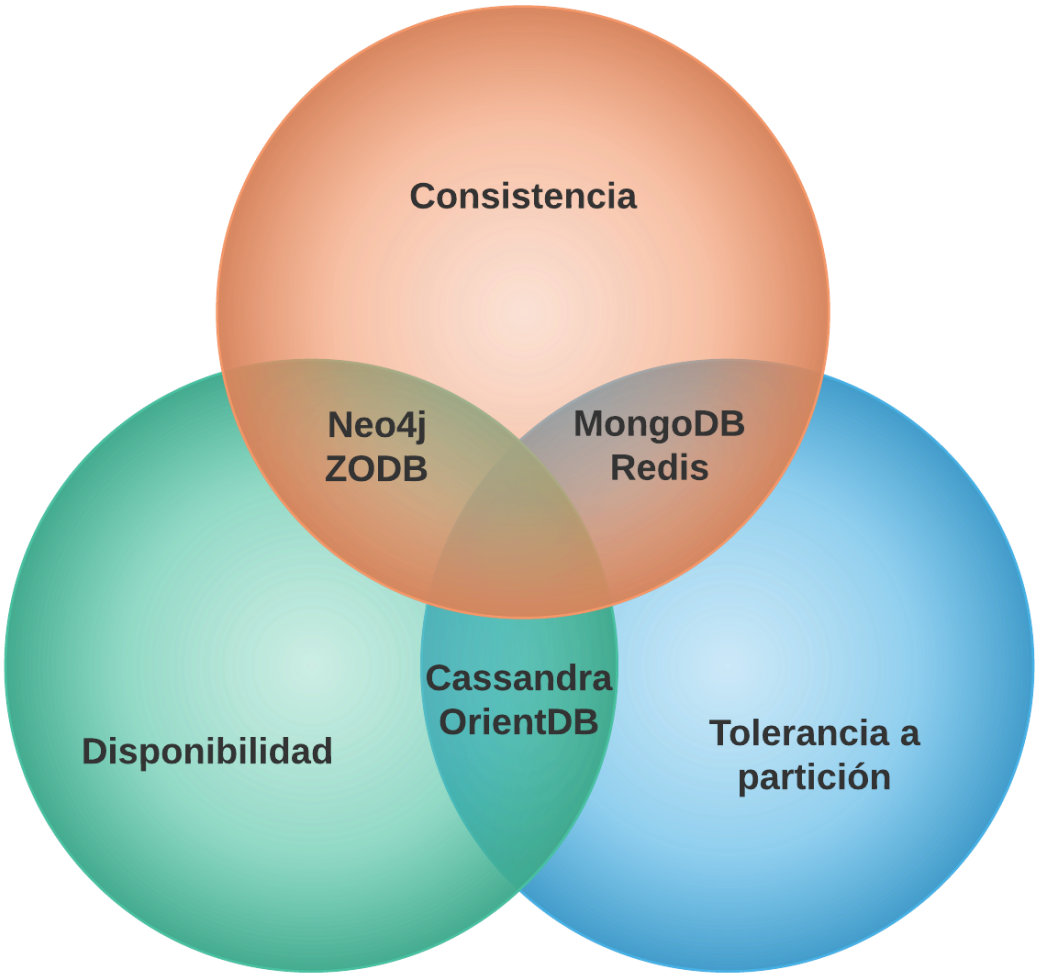


Figura 4.1 Tecnologías en Teorema CAP[11], configuraciones por defecto.

En la figura 4.1 se muestran dónde se encuentran las seis tecnologías dentro del Teorema CAP (Apéndice 1) con sus configuraciones por defecto, y se observa un equilibrio entre las tres características del teorema. La mayoría de las tecnologías aquí seleccionadas cuentan con herramientas para el ajuste de las características. Por ejemplo, MongoDB es capaz de ajustar su nivel de consistencia para asegurar la disponibilidad de múltiples escritores en la base de datos. Cassandra por su parte, es capaz de definir la cantidad de réplicas al escribir deben tener los datos antes de poder ser consultados, con poca cantidad pierde consistencia y gana disponibilidad y a mayor cantidad de réplicas necesarias mayor consistencia pero pierde disponibilidad.

[11] Consistency, Availability and Partition tolerance (Consistencia, Disponibilidad y tolerancia a particiones)

4.1 Bases de datos orientadas a documentos

Las bases de datos orientadas a documentos son vistas como un conjunto de archivos que guardan a su vez conjuntos de claves y valores o una sola clave y muchos valores con tipos de datos diversificados, generalmente representados en formatos semi estructurados como JSON, XML, BSON, YAML. Esto da como resultado que estas bases de datos sean generalmente multipropósito y que contengan grado alto de flexibilidad en los datos. Los sistemas proponen pocas limitaciones en la entrada de los datos, siempre y cuando mantengan la expresividad y lectura del documento, las diferencias entre manejadores de esta índole suelen ser la forma de consulta a los documentos, como la utilización de índices, *mapreducers*[12], así como la forma de replicación y consistencia de los datos.

4.1.1 MongoDB

MongoDB [38] es una base de datos libre orientada a documentos, fue diseñado pensando en una base de datos relacional como MySQL implementando diferente esquema de modelo de datos, buscando facilitar la escalabilidad horizontal en la capa de datos. Adopta la filosofía ágil de desarrollo, basándose en el modelo de documentos JSON/BSON para una fácil integración, un esquema flexible y buen desempeño al juntar información en un sólo lugar.

Nombre	MongoDB
Desarrollador	10gen
Replicación	Maestro-Esclavo
Partición	Sharding
Transacciones	No
MapReduce	Sí
Lenguajes soportados	C++, C, C#, Python, Ruby, PHP, Perl, Java, Scala, Haskell, Erlang
Lenguaje de consulta	Javascript
Teorema CAP	CP (Consistencia y Tolerancia a Partición)

Tabla 4.1.1.1 Características de MongoDB

[12] Modelo utilizado para soportar el cómputo distribuido

4.2 Bases de datos orientadas a columnas

Su diseño radica en que el diseño de los datos se agrupan en columnas, a diferencia de filas como en las bases de datos relacionales, esto ayuda a que agregar columnas sea de forma sencilla y natural. Cada fila puede contener diferentes columnas, incluso ninguna, evitando los valores nulos y logrando un administrador sin esquemas fijos.

4.2.1 Cassandra

Cassandra [39] es una base de datos basada en columnas, diseñada para que sea distribuida, con más de cinco años es una de las bases de datos más maduras y estables. El modelado de los datos depende de las consultas requeridas. Ya que es distribuida, aplica diversas estrategias para el conocimiento de nuevos nodos y la determinación de las particiones de información en cada uno de los nodos. Cassandra inicialmente asegura la disponibilidad, sin embargo, cuenta con una herramienta para dar mayor prioridad a la consistencia de los datos.

Nombre	Cassandra
Desarrollador	Apache Software Foundation
Replicación	Multi-Maestro
Partición	Random partitioner
Transacciones	Recientemente soporta transacciones ligeras
MapReduce	Sí
Lenguajes soportados	Python, Java, Node.js, Clojure, .NET, Ruby, PHP, Perl, Go, Haskell, C++
Lenguaje de consulta	CQL
Teorema CAP	AP (Disponibilidad y Tolerancia a Partición)

Tabla 4.2.1.1 Características de Cassandra

4.3 Bases de datos multimodelo

En la actualidad varios proyectos tratan de juntar los beneficios de dos y hasta tres tecnologías tanto en la forma de almacenamiento como en las formas que se pueda consultar la información. Existen diferentes combinaciones de almacenamiento, las más comunes son clave-valor y orientadas a columnas, documentos y grafos, documentos y relacional, entre otras con módulos intercambiables para las consultas y la forma de almacenamiento. Es el tipo de NoSQL y NewSQL[13] con más crecimiento.

4.3.1 OrientDB

OrientDB [40] es un manejador de base de datos multimodelo, que soporta grafos y es orientada a documentos. Aún utilizando documentos, las relaciones entre ellos son manejadas como grafos, con diferentes niveles de limitaciones a esquemas y soporta la utilización de usuarios y roles que varias NoSQL no incluyen como seguridad. Además tiene soporte para SQL.

Nombre	OrientDB
Desarrollador	OrienTechnologies
Replicación	Multi-Maestro
Partición	Sí
Transacciones	Sí
MapReduce	No
Lenguajes soportados	Java, Javascript, Node.js, Scala, C, PHP, Ruby, .NET, Python, Clojure
Lenguaje de consulta	Gremlin, SQL*
Teorema CAP	AP (Disponibilidad y Tolerancia a Partición)

*Soporta un subconjunto de operaciones

Tabla 4.3.1.1 Características de OrientDB

4.4 Bases de datos orientadas a grafos

Tiene como fundamento la estructura de datos de grafo, que se compone de nodos y relaciones entre los nodos (aristas), tanto los nodos como las relaciones pueden tener propiedades. los manejadores de bases de datos orientadas a grafos hacen énfasis en la navegación y búsqueda de la información deseada. Esto se puede contrastar con el enfoque de los manejadores relacionales, donde la búsqueda de la información es a través del filtrado de propiedades, buscando que cumplan con alguna característica deseada.

4.4.1 Neo4J

Neo4j [41] es una base de datos orientada a grafos nativa, soporta operaciones ACID[13], utiliza Cypher como lenguaje de operaciones, diseñado específicamente para realizar operaciones con grafos, se enfoca en la sintaxis de qué obtener del grafo, los esfuerzos de mejora del lenguaje se dirigen a la forma de leerlo no de escribirlo y toma algunos elementos de sintaxis de SQL.

Nombre	Neo4j
Desarrollador	OrienTechnologies
Replicación	Maestro-esclavo
Partición	No
Transacciones	Sí
MapReduce	No
Lenguajes soportados	REST, Java, Ruby, PHP, .NET, Python, Node.js, Clojure, Scala, Perl, Go, Haskell
Lenguaje de consulta	Gremlin, Cypher
Teorema CAP	CA (Consistencia y Disponibilidad)

Tabla 4.4.1.1 Características de Neo4j

[13] Atomicidad, Consistencia, Aislamiento y Durabilidad (Atomicity, Consistency, Isolation, Durability)

4.5 Bases de datos Clave-Valor

Es el modelo más simple, el almacenamiento de claves y sus valores. Esta simplicidad se ve reflejada en un excelente desempeño, por lo que estas bases de datos son utilizadas para sistemas de tiempo real y sistemas donde la recuperación de datos en poco tiempo es esencial. Si se requieren consultas complejas, es común que se guarden tipos de valores complejos como listas y se delegue el análisis de los datos a la implementación del sistema.

4.5.1 Redis

Redis [42] es una base de datos en memoria, es referido como servidor de estructura de datos, por los tipos de datos que soporta (hashes, listas, conjuntos, strings). En primera instancia guarda la información en memoria para luego hacerla persistente en disco después de algún tiempo o cantidad de consultas realizadas, ambas opciones son configurables conforme a las necesidades de la aplicación.

Nombre	Redis
Desarrollador	Salvatore Sanfilippo
Replicación	Maestro-Multi-Esclavo
Partición	Sí
Transacciones	Sí
MapReduce	No
Lenguajes soportados	C, C++, C#, Clojure, Lisp, Erlang, Go, Haskell, Haxe, io, Ruby, Lua, Objective-C, Perl, PHP, Python, Scala, Smalltalk
Lenguaje de consulta	Comandos Redis
Teorema CAP	CP (Consistencia y Tolerancia a Partición)

Tabla 4.5.1.1 Características de Redis

4.6 Bases de datos orientadas a objetos

El paradigma de programación orientado a objetos es el más utilizado en la actualidad. Para lograr que los objetos sean persistentes, en el ambiente relacional, se utilizan librerías y patrones de Mapeo Objeto-Relacional (ORM)[43]. Esta estrategia ocasiona problemas por la disparidad de paradigmas [44] como la baja de desempeño. Las bases de datos orientadas a objetos solucionan estos problemas al almacenar los objetos de manera nativa.

4.6.1 ZODB

ZODB [45] es una base de datos orientada a objetos nativa de Python, creando persistencia en los objetos incluyendo *savepoints* para lograr transacciones ACID. Además guarda el historial de los objetos, y la habilidad de deshacer los cambios guardados.

Nombre	ZODB
Desarrollador	Zope Corporation
Replicación	Maestro-Esclavo
Partición	No
Transacciones	Sí
MapReduce	No
Lenguajes soportados	Python
Lenguaje de consulta	Python
Teorema CAP	CA (Consistencia y Disponibilidad)

Tabla 4.6.1.1 Características de ZODB

En el Apéndice II se encuentra una tabla con las seis bases de datos aquí descritas y el resumen completo de las características. En el siguiente capítulo se muestra el experimento, iniciando con la implementación de cada modelo y los resultados obtenidos en el desempeño de las operaciones de escritura y lectura.

5 Experimento NoSQL

En este capítulo se describen los pasos realizados para la comparación de las seis tecnologías NoSQL elegidas para el experimento, desde el primer paso referente a la implementación, pasando por la medición y método para replicar el experimento.

5.1 Metodología

De acuerdo al experimento diseñado en el estudio realizado por Jayathilake [8], base de esta investigación, se evaluaron la creación de nodos y las consultas del árbol balanceado de grado 10 y una profundidad de 5. Los atributos agregados al árbol son tomados de acuerdo a un conjunto de datos mixtos como enteros, cadenas de caracteres, booleanos, etc. Se realizó la investigación siguiendo los siguientes pasos para cada una de las tecnologías:

1) Consulta para insertar 100,000 nodos al árbol.
2) Consulta para regresar un subárbol del nivel 2 de 1,000 nodos.
3) Consulta para regresar los nodos que contengan cierto atributo.
4) Consulta para regresar los nodos que contengan un valor específico a un atributo.

Para la presente investigación, se intenta replicar el experimento realizando todos los pasos de la comparación pero aplicados un problema específico (datos jerárquicos) con la información de códigos postales de México, descrita a detalle en la sección 5.2. Además se genera la información y los recursos necesarios para la replicación futura del experimento. Para ello se diseñaron los siguientes pasos, que se repitieron para cada una de las bases de datos NoSQL seleccionadas.

1) Selección del modelado de datos de la tecnología. Se investigan los diferentes tipos de modelados y alternativas que cada tecnología ofrece al acercarse al modelado de datos jerárquicos. La selección del modelado se realiza de acuerdo a los siguientes criterios:
    a) La utilización de la tecnología con configuraciones por defecto.
    b) Continuar con los modelados de datos sugeridos y desarrollados por la tecnología.
    c) El modelado que naturalmente se adapte al problema. Es importante rescatar aquellos que de forma sencilla representan una jerarquía y se adaptan a las consultas requeridas.
    d) El modelado que permita operaciones inherentes a los datos jerárquicos.

2) Implementación del modelado. Se realiza el modelado de la jerarquía. Se insertan en la base de datos jerarquía 107,630 nodos de los datos de la jerarquía geográfica de México (Q1).
3) Implementación de consultas:
    a) Encontrar un nodo por un identificador específico (Q2).
    b) Filtrar los nodos de acuerdo a un valor de atributo (Q3).
    c) Regresar una sub jerarquía (Q4).
4) Registro de los tiempos de ejecución (tiempos de CPU y tiempos de espera), en segundos, de las consultas. Los datos son mostrados en la sección 5.3.
5) Desarrollo de una máquina virtual por medio de vagrant[46], instalando cada administrador de base de datos NoSQL seleccionado, para asegurar la replicación y portabilidad del experimento.

5.2 Implementación

La jerarquía mostrada en la figura 5.2.1 fue desarrollada para modelar la información encontrada en GeoNames[47], se tomó únicamente la información relacionada a los códigos postales y no a la información referente a cada lugar. La implementación está diseñada para representar todas las localidades del mundo. Algunos países varían en el número de divisiones administrativas. Las implementaciones durante el experimento fueron realizadas únicamente sobre los datos de México [48].

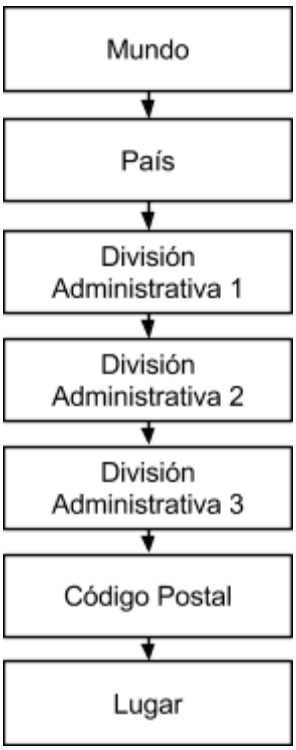


Figura 5.2.1 Jerarquía de GeoNames

La información proporcionada por GeoNames acerca de los códigos postales contienen la siguiente estructura [49]:

Campo	Descripción	Tipo de dato
country code	código del país (iso)	2 caracteres
postal code	código postal	varchar(20)
place name	nombre del lugar	varchar(180)
admin name1	1. nombre de subdivisión (estado)	varchar(100)
admin code1	1. código de subdivisión (estado)	varchar(20)
admin name2	2. nombre de subdivisión (municipio)	varchar(100)
admin code2	2. código de subdivisión (municipio)	varchar(20)
admin name3	3. nombre de subdivisión (comunidad)	varchar(100)
admin code3	3. código de subdivisión (comunidad)	varchar(20)
latitude	latitud estimada	(wgs84[14])
longitude	longitud estimada	(wgs84)
accuracy	precisión de latitud y longitud	int(1)

Tabla 5.2.1 Estructura de Geonames

La implementación realizada para el experimento contiene 1 país, 32 estados, 2,450 municipios, 621 comunidades, 29,323 códigos postales y 75,203 localidades o colonias, lo que da un total de 107,630 nodos de información geográfica de México. Dicho número es similar a los 100,000 nodos del experimento base.

La figura 5.2.2 muestra un árbol que es un extracto de toda la información de los códigos postales en México y sirve como referencia para mostrar la implementación realizada en cada una de las tecnologías.

[14] Sistema de coordenadas

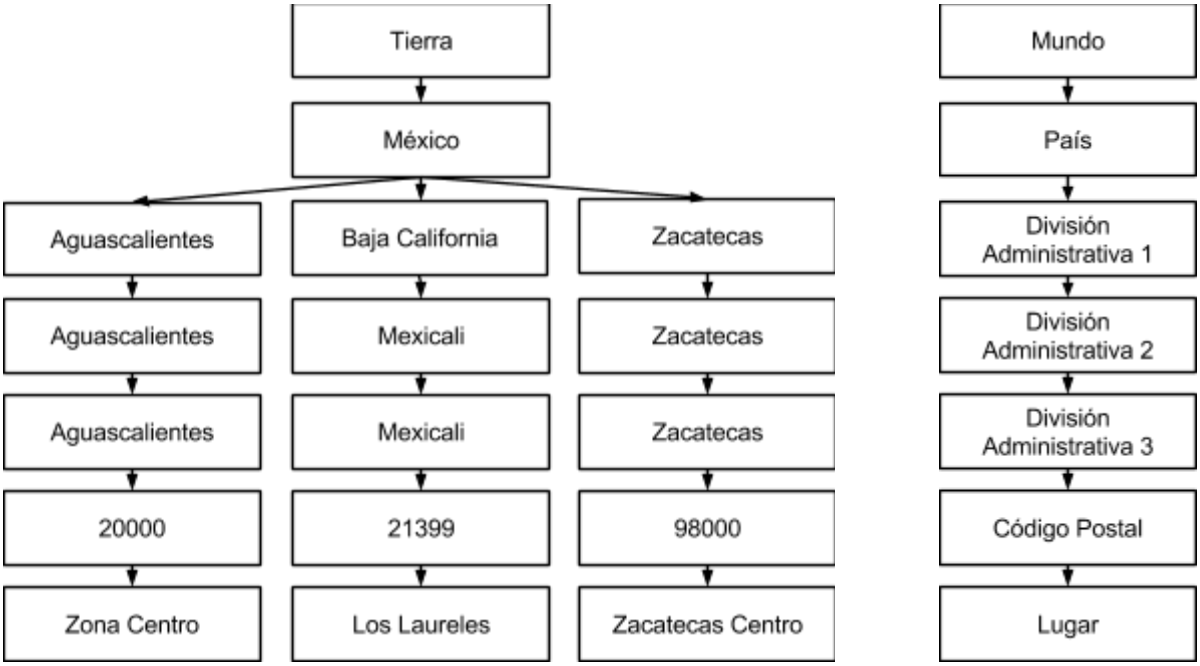


Figura 5.2.2 Extracto de información de la jerarquía

A continuación se muestran las diferentes implementaciones de las cuatro consultas evaluadas (Creación de nodos, consulta por id, consulta por valor de atributo y subjerarquía) en cada una de las tecnologías, utilizando como información muestra el extracto de la figura 5.2.2.

5.2.1 MongoDB (Orientado a documentos)

El modelado de datos en MongoDB se realiza a partir de documentos con estructura similar a JSON[15]. La base de datos puede contener diferentes colecciones que abarcan varios documentos, los documentos están compuestos por un conjunto de campos y sus valores (figura 5.2.1.2). Los datos tienen un esquema flexible, esto es, los documentos son estructuras flexibles, a pesar de estar dentro de una colección, no se obliga a contener campos ni valores específicos. Existen varias ventajas de utilizar documentos como forma de modelo de datos, una de ellas es la correspondencia de objetos y tipos de datos en lenguajes de programación.

[15] JavaScript Object Notation

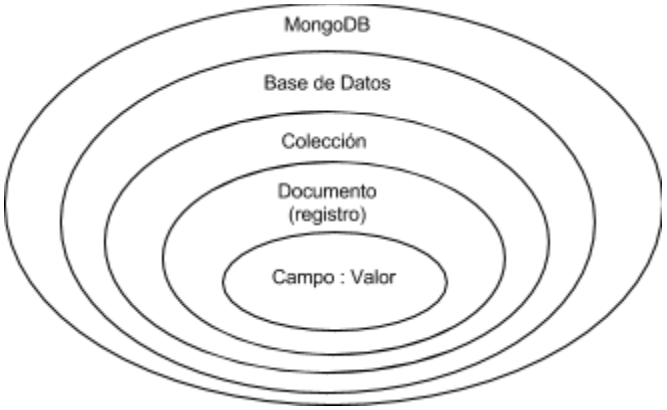


Figura 5.2.1.1 Modelo de MongoDB

Los registros en MongoDB son los documentos, contienen un identificador que los hace únicos dentro de una colección, las operaciones de escritura se realizan de forma atómica, a nivel de documentos, así que una operación de escritura no puede afectar más de un documento o más de una sola colección. En la figura 5.2.1.2 se muestra un ejemplo de un documento en MongoDB.

```
{
      "titulo": "100 años de soledad",
      "autor": "Gabriel García Márquez",
      "genero": ["novela", "realismo mágico"],
      "ISBN": "84-376-0494-X"
}
```

Figura 5.2.1.2 Documento en MongoDB

Las decisiones clave dentro del modelado se refieren al cómo representar relaciones entre los datos. Existen dos formas de representar dichas relaciones, la primera, a través de campos de referencia[1] y la segunda, a través de documentos anidados. Por ejemplo:

Representación de relaciones por referencia.

Documento: autor

```
{
      "_id": "1",
      "nombre": "Gabriel García Márquez",
      "años": "1927-2014",
      "pais": "Colombia"
}
```

Documento: libro

```
{
      "titulo":    "100    años    de
soledad",
      "autor_id": "1",
      "genero":  ["novela",  "realismo
mágico"],
      "ISBN": "84-376-0494-X"
}
```

Figura 5.2.1.3 Relaciones por referencia

Se incluyen referencias entre los documentos (Figura 5.2.1.3), las referencias normalmente son a través del identificador del documento, el campo "_id". Estas representaciones suelen realizar *n* cantidad de consultas de acuerdo al número de relaciones en un enfoque con datos normalizados.

Representación de relaciones por documentos incrustados.

Figura 5.2.1.4 Relaciones por subdocumentos

En la figura 5.2.1.4 se muestra un ejemplo donde el documento de autor es incrustado al documento del libro, es un enfoque desnormalizado donde los datos pueden repetirse y el subdocumento del autor es agregado a cada uno de sus libros. En una consulta es posible obtener información múltiple de manera menos costosa, ya que es posible el manejo de datos y sus relaciones en una sola operación.

5.2.1.1 Implementando la jerarquía

En el manual de MongoDB [10] se exponen cinco diferentes modelos para la representación de datos jerárquicos, se optó por implementar el modelo de path materializado (Figura 5.2.1.1.1), ya explicado en el Capítulo 3.

Estrategia de modelado elegida	Path materializado
Justificación	Utilización de expresiones regulares dentro de las consultas, mayor desempeño que lista de ancestros, facilidad de implementación
Otras alternativas	lista de ancestros, lista de adyacencia, conjuntos anidados

Figura 5.2.1.1.1 Estretegias de modelado en MongoDB

Para el extracto de información de la jerarquía (Figura 5.2.2) se implementó de la siguiente manera:

```
db.world.insert({"_id":"earth", "name":"Earth", "path":None})
db.world.insert({"_id":"MX", "name":"MX", "path":"earth"})

db.world.insert({"_id":"AGU", "name":"Aguascalientes", "path":"earth,MX"})
db.world.insert({"_id":"1001", "name":"Aguascalientes", "path":"earth,MX,AGU"})
db.world.insert({"_id":"CP20000", "name":"20000", "path":"earth,MX,AGU,1001"})
db.world.insert({"name":"Zona Centro", "path":"earth,MX,AGU,1001,CP20000"})

db.world.insert({"_id":"BCN", "name":"Baja California", "path":"earth,MX"})
db.world.insert({"_id":"2002", "name":"Mexicali", "path":"earth,MX,BCN"})
db.world.insert({"_id":"CP21399", "name":"Mexicali", "path":"earth,MX,BCN,2002"})
db.world.insert({"name":"Los Laureles", "path":"earth,MX,BCN,2002,CP21399"})

db.world.insert({"_id":"ZAC", "name":"Zacatecas", "path":"earth,MX"})
db.world.insert({"_id":"32056", "name":"Zacatecas", "path":"earth,MX,ZAC"})
db.world.insert({"_id":"CP98000", "name":"Zacatecas", "path":"earth,MX,ZAC,
32056"})
db.world.insert({"name":"Zacatecas Centro", "path":"earth,MX,ZAC,32056,CP98000"})
```

Figura 5.2.1.1.2 Extracto de la jerarquía implementada en MongoDB

Se utilizó el campo path para la concatenación de todos los ancestros del nodo separados por "," usualmente es utilizado el separador "/" pero, se optó por una coma para la simplificación en los caracteres especiales de las expresiones regulares. Se utilizaron los códigos de país, estado y municipio como identificadores y se agregó "CP" a todos los códigos postales para diferenciarlos de códigos de municipios. Para todos los lugares se dejó vacío el campo "_id", MongoDB le agrega un identificador específico.

5.2.1.2 Consultando la jerarquía

Como ya fue mencionado en el capítulo 3, el modelo de path materializado requiere operadores de texto. Las consultas fueron realizadas de acuerdo al manual de operaciones de lectura [50]. La consulta se realiza dentro de una colección de documentos a partir del método *db.collection.find()*, se envían los criterios de búsqueda así como proyecciones[52] y regresa un cursor[53]. Adicionalmente se pueden utilizar los modificadores limit(), skip() y sort() para ordenar, limitar y saltar los registros regresados en el cursor.

Regresando un nodo por su identificador:

```
db.world.findOne({"_id":"CP21399"})
```

Figura 5.2.1.2.1 Consulta por id (Q2)

El método anterior es una variación agregada por MongoDB para regresar un sólo documento.

Regresando los nodos por el valor de un atributo:

```
db.world.find({"name":"Aguascalientes"})
```

Figura 5.2.1.2.2 Consulta por valor de atributo (Q3)

Regresando la subjerarquía de un nodo:

```
db.world.find({"path":{"$regex":",32056"}})
```

Figura 5.2.1.2.3 Consulta de subjerarquía (Q4)

La consulta anterior utiliza expresiones regulares para encontrar el patrón “,32056” en el *path* de todos los ancestros, regresando todos los descendientes del municipio. La consulta no regresa el nodo raíz de la subjerarquía para ello es necesario agregar la consulta de nodo por identificador.

5.2.2 Cassandra (Orientado a columnas)

El modelo de columnas no es muy diferente a las tablas relacionales y el desarrollo de CQL[16] [54] se ha creado para que la transición entre un modelo relacional y cassandra se realice de una forma sencilla y de poco esfuerzo para aquellos que estén familiarizados con SQL. El modelado de Cassandra, como de muchas bases de datos NoSQL, depende más de las consultas que son requeridas, ya que es posible un esquema que va de semi flexible hasta completamente flexible.

Cassandra está diseñada para trabajar como un conjunto de nodos o cluster. Toda la información es particionada y replicada al conjunto de nodos. Existen dos conceptos clave que deben considerarse durante la implementación en Cassandra: 1) Dentro de un *Keyspace* se agregan y se configuran la forma en que se va a guardar la información y las estrategias de réplica y 2) dentro de las tablas se guardan los registros con la información (Figura 5.2.2.1). Los valores son guardados conforme a un nombre de columna, y su tiempo de modificado o inserción que sirve para definir las réplicas. Para Cassandra, la inserción y actualización de datos, es la misma operación.

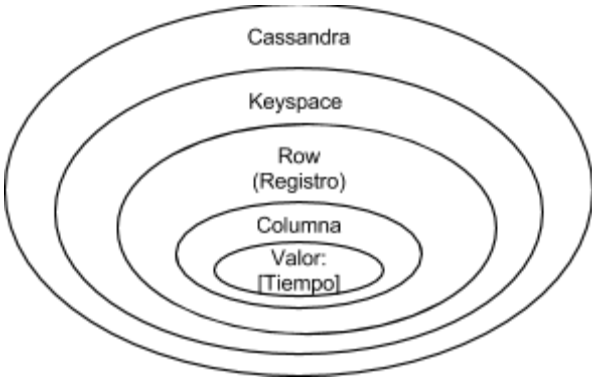


Figura 5.2.2.1 Modelo de Cassandra

En la figura 5.2.2.2 se muestra una implementación de datos denormalizados. A pesar de que se realizan más inserciones y se duplican datos, conforme a las consultas, en comparación con las tablas relacionales, las búsquedas se realizan con mayor eficiencia y desempeño, es por eso que el modelo fomenta el uso de datos denormalizados. Cassandra además utiliza un campo para partición o *partition key* y campos para ordenarlos llamados *cluster columns*. En el ejemplo se muestra una librería con diversas sucursales, utilizando llaves compuestas para la partición de la información y la creación de índices.

[16] Cassandra Query Language

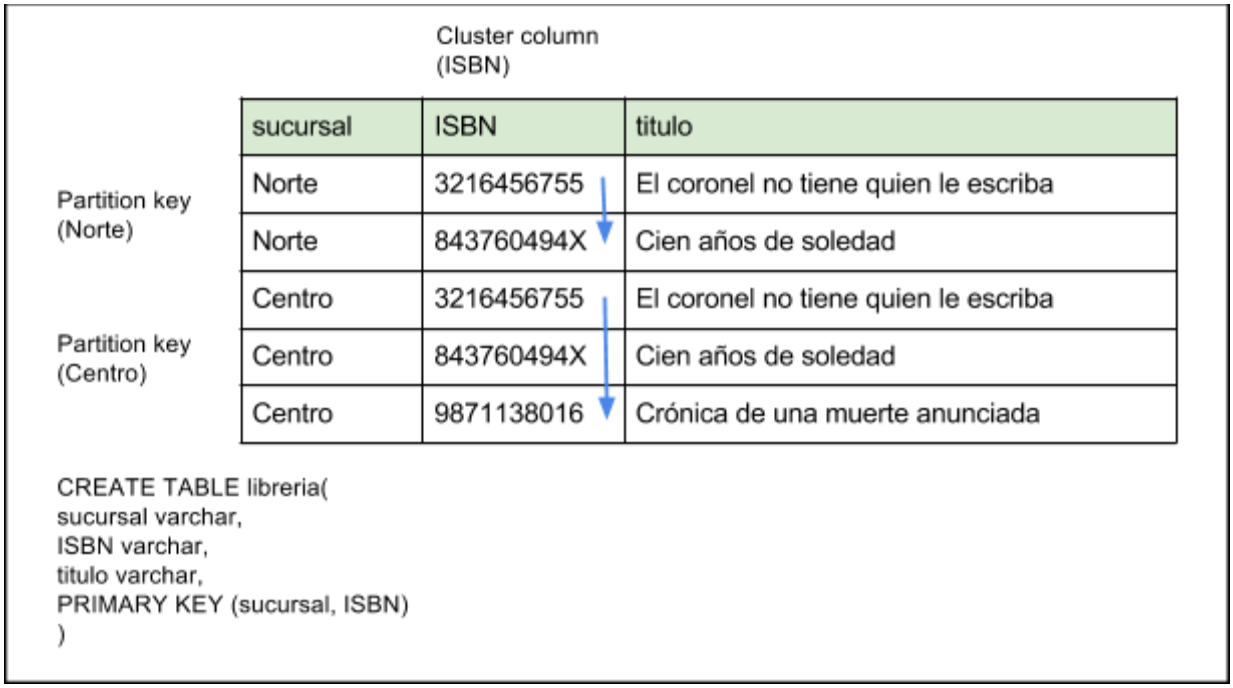


Figura 5.2.2.2 Tabla en Cassandra con llaves compuestas

5.2.2.1 Implementando la jerarquía

Las estrategias de representación en Cassandra se limita a utilizar campos como referencias a otros registros, anteriormente eran utilizadas las super columnas [55]. Se podrían implementar los modelos estudiados y propuestos para las bases de datos relacionales en el Capítulo 3. Sin embargo se optó por la lista de ancestros (Figura 5.2.2.1.1).

A partir de la versión 1.2 de Cassandra las colecciones son soportadas como tipos de datos [56], lo que ayuda al modelado de datos y existe una buena correlación entre el modelado y los lenguajes de programación. Las listas, como parte de estas colecciones, y sus métodos, ayudan a entender una jerarquía, es por esto que se decidió utilizar un modelado que se deriva del path materializado: la lista de ancestros[50].

Estrategia de modelado elegida	Lista de ancestros
Justificación	Aún no se cuenta con expresiones regulares dentro de las consultas pero se cuenta con operaciones de listas, fácil implementación.
Otras alternativas	path materializado, lista de adyacencia, conjuntos anidados

Figura 5.2.2.1.1 Estrategias de modelado en Cassandra

En la figura 5.2.1.1.2 se muestra el esquema utilizado para la implementación realizada en la herramienta cqlsh[57] integrada en Cassandra, un esquema en el cual sólo se utiliza un nodo servidor dentro del cluster y una tabla 'mexico' donde se guarda la información de cada nodo en la jerarquía. Se utiliza el campo 'id' utilizado para partición y 'name' para agrupación de la información, además de un campo de identificador, un nombre del nodo de la jerarquía, una lista con los ancestros y una lista para el tipo de nodo.

```
CREATE KEYSPACE list_path WITH replication =
{'class':'SimpleStrategy', 'replication_factor':1} ;

USE list_path;

CREATE TABLE mexico (id text, name text, ancestors list<text>, tags
list<text>, PRIMARY KEY(id, name));
```

Figura 5.2.2.1.2 Esquema en Cassandra

En una lista de ancestros se agrega cada uno de los ancestros a un campo de tipo lista y las consultas y modificaciones a la jerarquía se realizan a partir de las operaciones a la lista. Se utilizó la versión 2.1 de Cassandra, ya que cuenta con las operaciones sobre listas, necesarias para el experimento.

```
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('earth', 'Earth', [],
['world'] );

INSERT INTO mexico (id, name, ancestors, tags) VALUES ('MX', 'MX', ['earth'],
['country'] );

INSERT INTO mexico (id, name, ancestors, tags) VALUES ('AGU', 'Aguascalientes',
['earth', 'MX'], ['admin1'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('1001', 'Aguascalientes',
['earth', 'MX', 'AGU'], ['admin2'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('Aguascalientes',
'Aguascalientes', ['earth', 'MX', 'AGU', '1001'], ['admin3'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('CP20000', '20000',
['earth', 'MX', 'AGU', '1001', 'Aguascalientes'], ['code'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('P1', 'Zona Centro',
['earth', 'MX', 'AGU', '1001', 'Aguascalientes', 'CP20000'], ['place'] );

INSERT INTO mexico (id, name, ancestors, tags) VALUES ('BCN', 'Baja California',
['earth', 'MX'], ['admin1'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('2002', 'Mexicali',
['earth', 'MX', 'BCN'], ['admin2'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('Mexicali', 'Mexicali',
['earth', 'MX', 'BCN', '2002'], ['admin3'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('CP21399', '21399',
['earth', 'MX', 'BCN', '2002', 'Mexicali'], ['code'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('P1782', 'Los Laureles',
['earth', 'MX', 'BCN', '2002', 'Mexicali', 'CP21399'], ['place'] );

INSERT INTO mexico (id, name, ancestors, tags) VALUES ('ZAC', 'Zacatecas',
['earth', 'MX'], ['admin1'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('32056', 'Zacatecas',
['earth', 'MX', 'ZAC'], ['admin2'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('Zacatecas', 'Zacatecas',
['earth', 'MX', 'ZAC', '32056'], ['admin3'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('CP98000', '98000',
['earth', 'MX', 'ZAC', '32056', 'Zacatecas'], ['code'] );
INSERT INTO mexico (id, name, ancestors, tags) VALUES ('P75004', 'Zacatecas
Centro', ['earth', 'MX', 'ZAC', '32056', 'Zacatecas', 'CP98000'], ['place'] );
```

Figura 5.2.2.1.3 Extracto de la jerarquía implementada en Cassandra

5.2.2.2 Consultando la jerarquía

Durante la implementación en Cassandra (Figura 5.2.2.1.3) se utilizó el tipo de datos de listas [58]. Además, se deben crear índices para aquellos campos en los que se requiera realizar una búsqueda.

Algunas comparaciones muestran un cambio en el desempeño de Cassandra al agregar más nodos al cluster de información [59].

En la figura 5.2.2.2.1 se crean los ndices necesarios para realizar consultas sobre dos columnas (ancestros y tags).

```
USE list_path;

CREATE INDEX mex_path ON mexico(ancestors);

CREATE INDEX mex_tags ON mexico(tags);
```

Figura 5.2.2.2.1 Agregar índices en Cassandra

Regresando un nodo por su identificador:

```
SELECT * FROM mexico WHERE id = 'CP21399' limit 1;
```

Figura 5.2.2.2.2 Consulta por id (Q2)

Se utiliza el parámetro limit para detener la búsqueda al encontrar el registro.

Regresando los nodos por el valor de un atributo:

```
SELECT * FROM mexico WHERE name = 'Aguascalientes' ALLOW FILTERING;
```

Figura 5.2.2.2.3 Consulta por valor de atributo (Q3)

Ya que regresamos toda la información del nodo, agregamos el predicado de 'ALLOW FILTERING' para indicar que la consulta requiere un filtrado por el valor del atributo 'name'.

Regresando la subjerarquía de un nodo:

```
SELECT * FROM mexico WHERE ancestors contains '32056';
```

Figura 5.2.2.2.4 Consulta de subjerarquía (Q4)

La consulta anterior utiliza el método *contains* agregado en la versión 2.1 beta [60], en donde a partir del índice creado para los ancestros se busca que se encuentre en la lista el ancestro "32056", regresando todos los descendientes del municipio, 224 nodos. La consulta no regresa el nodo raíz de la subjerarquía para ello es necesario agregar la consulta de nodo por identificador.

5.2.3 OrientDB (Multimodelo)

Las implementaciones de las bases de datos que soportan grafos son naturales. La forma en que se representa la estructura de la información y la implementación es básicamente la misma. OrientDB es capaz de utilizar el modelo en base a documentos y en base a grafos, en la figura 5.2.3.1 muestra los conceptos y la forma que se relacionan al utilizar alguno de los dos modelos.

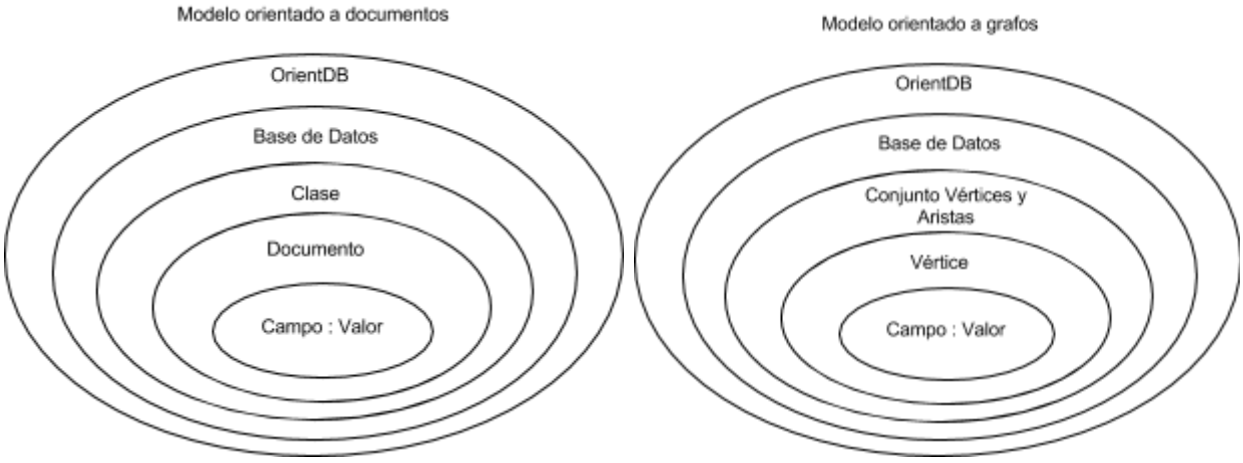


Figura 5.2.3.1 Modelos en OrientDB

OrientDB trabaja bajo un modelado sin esquema fijo, esto es que se pueden agregar las propiedades necesarias dentro de la implementación conforme se va avanzando, pero también incluye clases para agregar propiedades, restricciones e índices; lo que agrega flexibilidad para proyectos con diferentes niveles de esquemas.

Para el experimento por la naturaleza del problema se utilizó el modelo orientado a grafos (Figura 5.2.3.2). El modelo consta principalmente del uso de dos clases ya definidas en los esquemas por defecto de la base de datos, la clase Vertex (V) para definir vértices como entidades independientes y a las cuales se les agregan propiedades y valores, y la clase Edge (E) para describir las relaciones entre los vértices.

Estrategia de modelado elegida	Grafo
Justificación	Estructura del problema es la misma que el modelado de los datos
Otras alternativas	lista de ancestros, lista de adyacencia, conjuntos anidados, path materializado

Figura 5.2.3.2 Estrategias de modelado en OrientDB

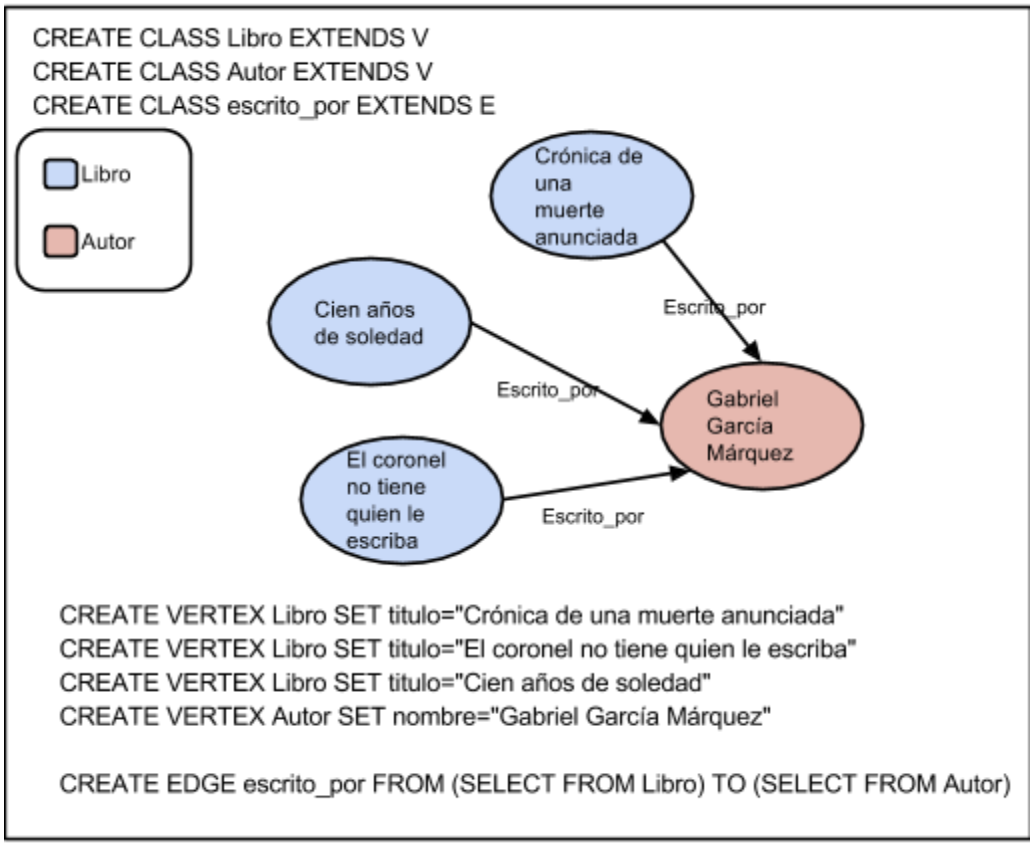


Figura 5.2.3.3 Ejemplo de vértices y aristas

En la figura 5.2.3.3 se muestra un ejemplo de implementación en los comandos de OrientDB que se asemejan a la utilización de SQL [61]. Los primeros tres comandos de la figura, se refieren al esquema utilizado, es posible crear propiedades y restricciones para hacer respetar los atributos de cada nodo, los comandos *CREATE VERTEX* y *CREATE EDGE* son utilizados para el manejo de grafos. La creación de aristas se realiza en una sola consulta, uniendo todos los libros con todos los actores, en este caso existe un solo autor y tres libros, para casos más específicos es necesario utilizar la cláusula *WHERE* para el filtrado de los nodos.

5.2.3.1 Implementando la jerarquía

A pesar de que en OrientDB utiliza esquemas flexibles, es una buena práctica crear clases y propiedades con la información que ya es conocida y agregar dinámicamente propiedades e información nueva, para ello se genera el esquema de la figura 5.2.3.1.1.

```
CREATE CLASS Admin1 extends V
CREATE PROPERTY Admin1.name string

CREATE CLASS Admin2 extends V
CREATE PROPERTY Admin2.name string

CREATE CLASS Admin3 extends V
CREATE PROPERTY Admin3.name string

CREATE CLASS Code extends V
CREATE PROPERTY Code.name string

CREATE CLASS Country extends V
CREATE PROPERTY Country.name string

CREATE CLASS Place extends V
CREATE PROPERTY Place.name string

CREATE CLASS World extends V
CREATE PROPERTY World.name string

CREATE CLASS Division extends E
```

Figura 5.2.3.1.1 Esquema en OrientDB

Una vez que se genera el esquema a partir de subclases de V (Vértices) y E (Aristas), es posible la creación del grafo. En el ejemplo de la figura 5.2.3.1.2 se muestra un extracto de la información geográfica de México, las aristas a diferencia de la figura 5.2.3.1 en las que se realizan a partir de consultas, se utilizan los identificadores agregados automáticamente por OrientDB. Este indicador se compone del número de la clase, seguido de un consecutivo.

```
CREATE VERTEX World SET name = 'Earth'

CREATE VERTEX Country SET name = 'MX'
CREATE EDGE Division FROM #20:0 TO #18:0

CREATE VERTEX Admin1 SET name = 'Aguascalientes'
CREATE EDGE Division FROM #18:0 TO #14:0
CREATE VERTEX Admin2 SET name = 'Aguascalientes'
CREATE EDGE Division FROM #14:0 TO #15:0
CREATE VERTEX Admin3 SET name = 'Aguascalientes'
CREATE EDGE Division FROM #15:0 TO #16:0
CREATE VERTEX Code SET name = '20000'
CREATE EDGE Division FROM #16:0 TO #17:0
CREATE VERTEX Place SET name = 'Zona Centro'
CREATE EDGE Division FROM #17:0 TO #19:0

CREATE VERTEX Admin1 SET name = 'Baja California'
CREATE EDGE Division FROM #18:0 TO #14:1
CREATE VERTEX Admin2 SET name = 'Mexicali'
CREATE EDGE Division FROM #14:1 TO #15:1
CREATE VERTEX Admin3 SET name = 'Mexicali'
CREATE EDGE Division FROM #15:1 TO #16:1
CREATE VERTEX Code SET name = '21399'
CREATE EDGE Division FROM #16:1 TO #17:1
CREATE VERTEX Place SET name = 'Los Laureles'
CREATE EDGE Division FROM #17:1 TO #19:1

CREATE VERTEX Admin1 SET name = 'Zacatecas'
CREATE EDGE Division FROM #18:0 TO #14:2
CREATE VERTEX Admin2 SET name = 'Zacatecas'
CREATE EDGE Division FROM #14:2 TO #15:2
CREATE VERTEX Admin3 SET name = 'Zacatecas'
CREATE EDGE Division FROM #15:2 TO #16:2
CREATE VERTEX Code SET name = '98000'
CREATE EDGE Division FROM #16:2 TO #17:2
CREATE VERTEX Place SET name = 'Zacatecas Centro'
CREATE EDGE Division FROM #17:2 TO #19:2
```

Figura 5.2.3.1.2 Extracto de la jerarquía implementada en OrientDB

5.2.3.2 Consultando la jerarquía

Las consultas se realizaron a partir de los comandos SQL de lectura, existen diferencias entre el estándar SQL-92 y el utilizado en OrientDB [62]. Se utilizó *SELECT* para regresar los nodos y *TRAVERSE* para realizar búsquedas a lo largo del grafo.

Regresando un nodo por su identificador:

```
SELECT FROM [#17:1]
```

Figura 5.2.3.2.1 Consulta por id (Q2)

La consulta regresa un sólo nodo por el id generado por OrientDB llamado RID(#clase:nodo), en este caso regresa de la clase 17(Code) el nodo con índice 1 (Código 21399).

Regresando los nodos por el valor de un atributo:

```
SELECT FROM V WHERE name='Aguascalientes'
```

Figura 5.2.3.2.2 Consulta por valor de atributo (Q3)

Las consultas permitidas son únicamente aplicadas directamente a una clase, para realizar una consulta sobre diversas clases es necesario realizar subconsultas y luego unirlas. Al utilizar el esquema de grafos, todos las clases vértice son una subclase de la clase V, por lo que se puede realizar una consulta sobre todos los vértices con un mismo atributo como en la figura 5.2.3.2.2.

Regresando la subjerarquía de un nodo:

```
TRAVERSE out_Division FROM (SELECT FROM Admin2 WHERE
name='Zacatecas') WHILE $depth <= 3
```

Figura 5.2.3.2.3 Consulta de subjerarquía (Q4)

Se utilizó el comando TRAVERSE para navegar dentro del grafo, siendo exclusivo para el modelo, y agregando una condición de la profundidad del grafo, se consultó el municipio de Zacatecas y a partir de ahí se realizó la operación hacia los nodos entrantes con conexión al nodo raíz.

5.2.4 Neo4j (Orientada a grafos)

Neo4j ha realizado grandes esfuerzos para convertir la naturalidad y ventajas de una representación de grafos hacia una base de datos, para que su creación y su consulta, se asemejen a estar trabajando directamente con grafos. Es por eso que Neo4j utiliza la estructura de la figura 5.2.4.1 en donde únicamente se guardan grafos y las consultas se realizan a través de los nodos y saltando a los nodos que contenga alguna relación.

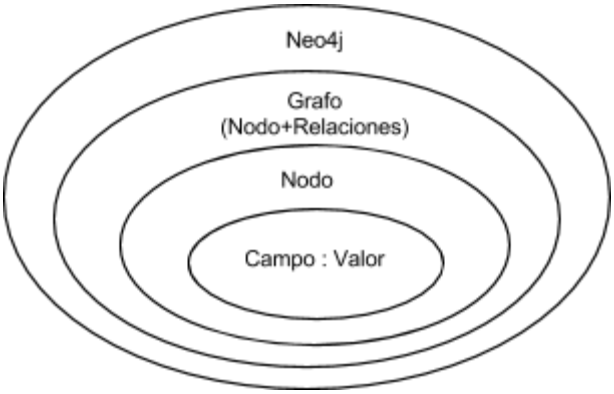


Figura 5.2.4.1 Modelo de Neo4j

Las consultas suelen regresar nodos, patrones de alguna estructura específica o caminos que indiquen la forma de llegar de un nodo a otro.

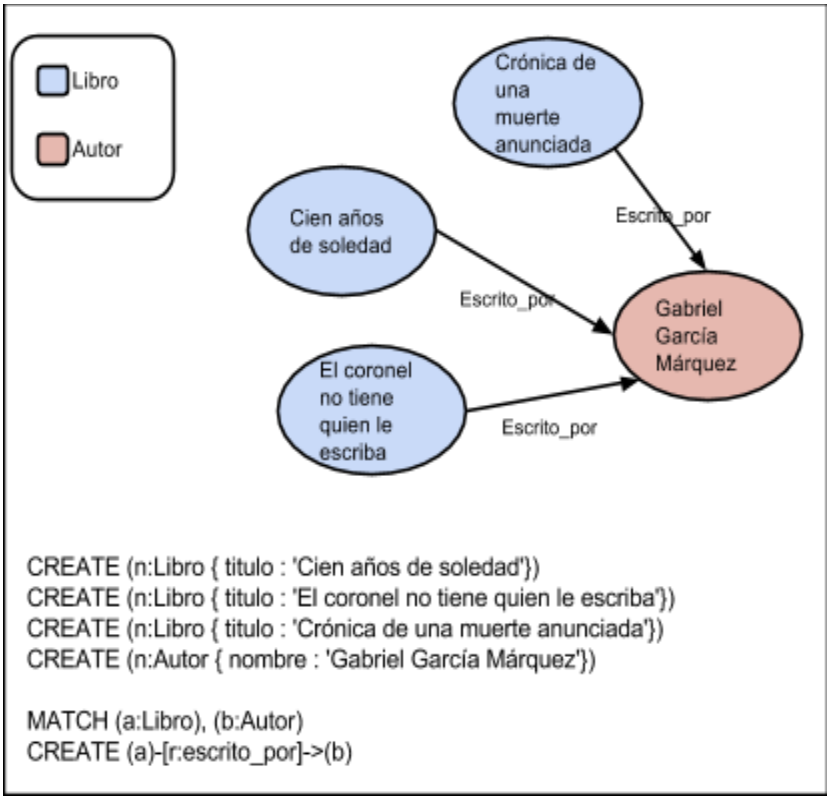


Figura 5.2.4.2 Ejemplo en Cypher de un grafo en Neo4j

5.2.4.1 Implementando la jerarquía

Como es mostrado en la Figura 5.2.4.1.1 es posible crear nodos y relaciones de diferentes formas. Al inicio se crearon dos nodos y se asignan a dos referencias, para después crear la relación entre los dos nodos. En el caso de los estados se crearon *paths* completos y al final fueron agregados al nodo país. Para poder retomar el nodo y utilizarlo en consultas subsecuentes agregamos el comando WITH después de la consulta que agrega el nodo.

```
CREATE (w:World {name: 'Earth'})
CREATE (c:Country {name: 'MX'})

WITH w, c
CREATE (w)-[r:Division]->(c)
WITH c

CREATE AGU =(a1:Admin1 { name:'Aguascalientes' })-[:Division]->
(b1:Admin2 { name:'Aguascalientes' })-[:Division]->
(c1:Admin3 { name:'Aguascalientes' })-[:Division]->
(d1:Code { name:'20000' })-[:Division]->
(e1:Place { name:'Zona Centro' })

WITH a1, c
CREATE (c)-[r:Division]->(a1)
CREATE BCN =(a2:Admin1 { name:'Baja California' })-[:Division]->
(b2:Admin2 { name:'Mexicali' })-[:Division]->
(c2:Admin3 { name:'Mexicali' })-[:Division]->
(d2:Code { name:'21399' })-[:Division]->
(e2:Place { name:'Los Laureles' })

WITH a2, c
CREATE (c)-[r:Division]->(a2)
CREATE ZAC =(a3:Admin1 { name:'Zacatecas' })-[:Division]->
(b3:Admin2 { name:'Zacatecas' })-[:Division]->
(c3:Admin3 { name:'Zacatecas' })-[:Division]->
(d3:Code { name:'98000' })-[:Division]->
(e3:Place { name:'Zacatecas Centro' })

WITH a3, c
CREATE (c)-[r:Division]->(a3)
```

Figura 5.2.4.1.1 Esquema en Neo4j

5.2.4.2 Consultando la jerarquía

Al ejecutar el servicio de Neo4j y la interfaz de servicio (http://localhost:7474), es posible realizar consultas en Cypher[17] [63] directamente y visualizar el grafo, a continuación se muestran las consultas desarrolladas para la investigación.

Regresando un nodo por su identificador:

```
START n=node(41) return n
```

Figura 5.2.4.2.1 Consulta por id (Q2)

Al agregar el comando START es posible iniciar una consulta a partir de un nodo determinado, en este caso se utilizó el identificador del nodo de código postal 21399. Neo4j agrega un identificador entero autoincremental a cada uno de los nodos.

Regresando los nodos por el valor de un atributo:

```
MATCH (n) WHERE n.name="Aguascalientes" RETURN n LIMIT 100
```

Figura 5.2.4.2.2 Consulta por valor de atributo (Q3)

Regresando la subjerarquía de un nodo:

```
MATCH p=(n:Admin2)-[*]->(m:Place) WHERE n.name="Zacatecas" return
p
```

Figura 5.2.4.2.3 Consulta de subjerarquía (Q4)

El comando MATCH es utilizado para regresar patrones en el grafo. En el caso de la figura 5.2.4.2.3 el patrón consta de dos nodos entre el tipo región administrativa 2 y el lugar, y es posible que se encuentren más relaciones y nodos entre ellos. al asignarlo a una variable, en este caso ‘p’, es posible regresar todo el path encontrado en el patrón.

[17] Lenguaje utilizado para la creación y modificación de grafos en Neo4j

5.2.5 Redis (Clave - Valor)

La cantidad de elementos en una estructura de una base de datos clave-valor es mínima en comparación a los demás modelos. En la figura 5.2.5.1 se muestra el modelo que utiliza Redis, pero hay que mencionar que existen diversos tipos de valores soportados y operaciones exclusivas para cada tipo de valor. Es posible concatenar cadenas de caracteres a el tipo específico en Redis o es posible obtener el mayor elemento dentro de un conjunto ordenado.

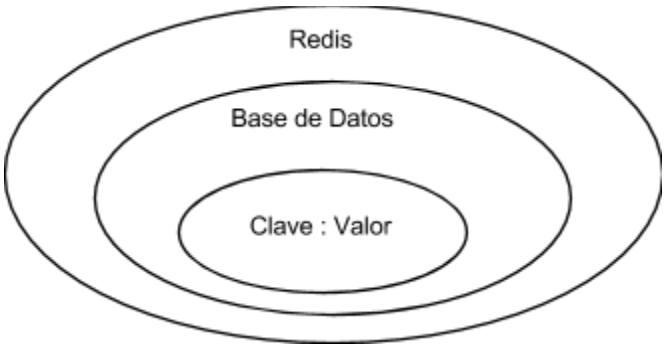


Figura 5.2.5.1 Modelo de Redis

El modelo en Redis es el más simple de los descritos en esta investigación, depende completamente del problema y de las consultas que se requieran y de la forma en que se ajustan a los tipos de datos de Redis. Redis cuenta con cinco tipos de datos, los strings es el valor más básico, las listas guardan un conjunto de strings, se utilizan cuando es necesario conocer el orden de los valores, de lo contrario se utiliza el tipo set, el hash se utiliza para crear un mapa entre las llaves y los valores. Por último los conjuntos ordenados o *sorted sets* que están vinculados a un marcador para poder ordenarlos.

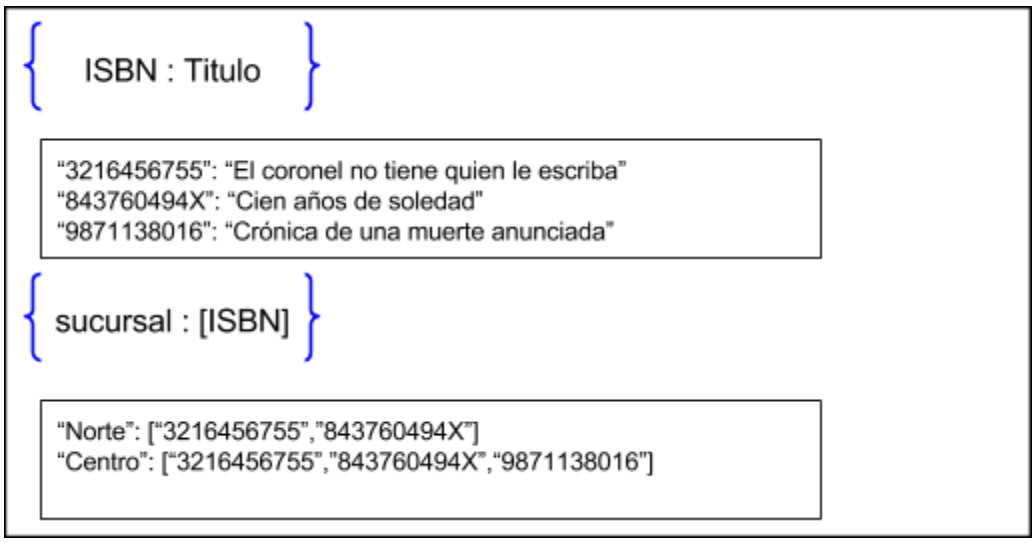


Figura 5.2.5.2 Ejemplo del modelo de libros en Redis

5.2.5.1 Implementando la jerarquía

Dado la simpleza del modelo de las bases de datos clave-valor, existen pocos modelos para representar jerarquías que sean manejadas dentro del administrador, se evaluó la utilización de los tipos de datos de listas y conjuntos, pero las limitaciones en los comandos relacionados a estos tipos de datos, originó que se decidiera por otro modelo no descrito en los capítulos anteriores: lista de tuplas, y tomado de la implementación en bases de datos relacionales de *closure tables* [64].

```
SET 'earth 'Earth'

SET 'MX' 'MX'
ZADD 'desc:earth'' 1 'MX'
ZADD 'asc:MX'' 1 'earth'

SET 'AGU' 'Aguascalientes'
ZADD 'desc:earth' 2 'AGU'
ZADD 'desc:MX' 1 'AGU'
ZADD 'asc:AGU' 1 'MX'
ZADD 'asc:AGU' 2 'earth'

SET '1001' 'Aguascalientes'
ZADD 'desc:earth' 3 'AGU'
ZADD 'desc:MX' 2 'AGU'
ZADD 'desc:AGU' 1 '1001'
ZADD 'asc:1001' 1 'AGU'
ZADD 'asc:1001' 2 'MX'
ZADD 'asc:1001' 3 'earth'

SET 'Aguascalientes' 'Aguascalientes'
ZADD 'desc:earth' 4 'AGU'
ZADD 'desc:MX' 3 'AGU'
ZADD 'desc:AGU' 2 '1001'
ZADD 'desc:1001' 1 'Aguascalientes'
ZADD 'asc:Aguascalientes' 1 '1001'
ZADD 'asc:Aguascalientes' 2 'AGU'
ZADD 'asc:Aguascalientes' 3 'MX'
ZADD 'asc:Aguascalientes' 4 'earth'

SET 'CP20000' '20000'
ZADD 'desc:earth' 5 'AGU'
ZADD 'desc:MX' 4 'AGU'
ZADD 'desc:AGU' 3 '1001'
ZADD 'desc:1001' 2 'Aguascalientes'
ZADD 'desc:Aguascalientes' 1 'CP20000'
ZADD 'asc:CP20000' 1 'Aguascalientes'
ZADD 'asc:CP20000' 2 '1001'
ZADD 'asc:CP20000' 3 'AGU'
ZADD 'asc:CP20000' 4 'MX'
ZADD 'asc:CP20000' 5 'earth'

SET 'P1' 'Zona Centro'
ZADD 'desc:earth' 6 'AGU'
ZADD 'desc:MX' 5 'AGU'
ZADD 'desc:AGU' 4 '1001'
ZADD 'desc:1001' 3 'Aguascalientes'
ZADD 'desc:Aguascalientes' 2 'CP20000'
ZADD 'desc:CP20000' 1 'P1'
ZADD 'asc:P1' 1 'CP20000'
ZADD 'asc:P1' 2 'Aguascalientes'
ZADD 'asc:P1' 3 '1001'
ZADD 'asc:P1' 4 'AGU'
ZADD 'asc:P1' 5 'MX'
ZADD 'asc:P1' 6 'earth'

SET 'BCN' 'Baja California'
ZADD 'desc:earth' 2 'BCN'
ZADD 'desc:MX' 1 'BCN'
ZADD 'asc:BCN' 1 'MX'
ZADD 'asc:BCN' 2 'earth'

SET '2002' 'Mexicali'
ZADD 'desc:earth' 3 'BCN'
ZADD 'desc:MX' 2 'BCN'
ZADD 'desc:BCN' 1 '2002'
ZADD 'asc:2002' 1 'BCN'
ZADD 'asc:2002' 2 'MX'
ZADD 'asc:2002' 3 'earth'

SET 'Mexicali' 'Mexicali'
ZADD 'desc:earth' 4 'BCN'
ZADD 'desc:MX' 3 'BCN'
ZADD 'desc:BCN' 2 '2002'
ZADD 'desc:2002' 1 'Mexicali'
ZADD 'asc:Mexicali' 1 '2002'
ZADD 'asc:Mexicali' 2 'BCN'
ZADD 'asc:Mexicali' 3 'MX'
ZADD 'asc:Mexicali' 4 'earth'

SET 'CP21399' '21399'
ZADD 'desc:earth' 5 'BCN'
ZADD 'desc:MX' 4 'BCN'
ZADD 'desc:BCN' 3 '2002'
ZADD 'desc:2002' 2 'Baja California'
ZADD 'desc:Mexicali' 1 'CP21399'
ZADD 'asc:CP21399' 1 'Mexicali'
ZADD 'asc:CP21399' 2 '2002'
ZADD 'asc:CP21399' 3 'BCN'
ZADD 'asc:CP21399' 4 'MX'
ZADD 'asc:CP21399' 5 'earth'

SET 'P1782' 'Los Laureles'
ZADD 'desc:earth' 6 'BCN'
ZADD 'desc:MX' 5 'BCN'
ZADD 'desc:BCN' 4 '2002'
ZADD 'desc:2002' 3 'Baja California'
ZADD 'desc:Mexicali' 2 'CP21399'
ZADD 'desc:CP21399' 1 'P1782'
ZADD 'asc:P1782' 1 'CP21399'
ZADD 'asc:P1782' 2 'Mexicali'
ZADD 'asc:P1782' 3 '2002'
ZADD 'asc:P1782' 4 'BCN'
ZADD 'asc:P1782' 5 'MX'
ZADD 'asc:P1782' 6 'earth'

SET 'ZAC' 'Zacatecas'
ZADD 'desc:earth' 2 'ZAC'
ZADD 'desc:MX' 1 'ZAC'
ZADD 'asc:ZAC' 1 'MX'
ZADD 'asc:ZAC' 2 'earth'

SET '32056' 'Zacatecas'
ZADD 'desc:earth' 3 'ZAC'
ZADD 'desc:MX' 2 'ZAC'
ZADD 'desc:ZAC' 1 '32056'
ZADD 'asc:32056' 1 'ZAC'
ZADD 'asc:32056' 2 'MX'
ZADD 'asc:32056' 3 'earth'

SET 'Zacatecas' 'Zacatecas'
ZADD 'desc:earth' 4 'ZAC'
ZADD 'desc:MX' 3 'ZAC'
ZADD 'desc:ZAC' 2 '32056'
ZADD 'desc:32056' 1 'Zacatecas'
ZADD 'asc:Zacatecas' 1 '32056'
ZADD 'asc:Zacatecas' 2 'ZAC'
ZADD 'asc:Zacatecas' 3 'MX'
ZADD 'asc:Zacatecas' 4 'earth'

SET 'CP98000' '98000'
ZADD 'desc:earth' 5 'ZAC'
ZADD 'desc:MX' 4 'ZAC'
ZADD 'desc:ZAC' 3 '32056'
ZADD 'desc:32056' 2 'Zacatecas'
ZADD 'desc:Zacatecas' 1 'CP98000'
ZADD 'asc:CP98000' 1 'Zacatecas'
ZADD 'asc:CP98000' 2 '32056'
ZADD 'asc:CP98000' 3 'ZAC'
ZADD 'asc:CP98000' 4 'MX'
ZADD 'asc:CP98000' 5 'earth'

SET 'P75004' 'Zacatecas Centro'
ZADD 'desc:earth' 6 'ZAC'
ZADD 'desc:MX' 5 'ZAC'
ZADD 'desc:ZAC' 4 '32056'
ZADD 'desc:32056' 3 'Zacatecas'
ZADD 'desc:Zacatecas' 2 'CP98000'
ZADD 'desc:CP98000' 1 'P75004'
ZADD 'asc:P75004' 1 'CP98000'
ZADD 'asc:P75004' 2 'Zacatecas'
ZADD 'asc:P75004' 3 '32056'
ZADD 'asc:P75004' 4 'ZAC'
ZADD 'asc:P75004' 5 'MX'
ZADD 'asc:P75004' 6 'earth'
```

Figura 5.2.5.1.1 Esquema en Redis

La estrategia de *closure tables* se encarga de almacenar todos los caminos posibles dentro de la jerarquía, almacenando un par de identificadores de nodos y la profundidad que existe entre ellos. Para ello se utilizó el tipo de datos de conjuntos ordenados, en donde el marcador o score representa la distancia entre los nodos. Se utilizaron dos conjuntos para las consultas descendentes y ascendentes.

En la figura 5.2.5.1.1 se muestra la implementación del extracto de información, agregando el nodo, seguido de agregarlo a la lista de descendientes de sus ancestros y a su lista de ancestros para ambas consultas.

Estrategia de modelado elegida	Closure tables
Justificación	Las operaciones de escritura son de bajo costo, se obtienen consultas de profundidad y subjerarquías.
Otras alternativas	lista de adyacencia, path materializado

Figura 5.2.5.1.2 Estretegias de modelado en Redis

5.2.5.2 Consultando la jerarquía

Para las consultas se utilizaron comandos sobre los conjuntos ordenados[65].

Regresando un nodo por su identificador:

```
GET 'CP21399'
```

Figura 5.2.5.2.1 Consulta por id (Q2)

Para regresar el código postal 21399 por su llave, es la consulta más sencilla y eficiente, el modelo clave-valor fue diseñado para esta clase de consultas, a través del comando *GET*.

Regresando los nodos por el valor de un atributo:

Las consultas de búsquedas por un valor van en contra del modelo de la base de datos, por lo que no son soportadas, sin embargo, se puede desarrollar conforme a las necesidades de la consulta, cambiando el valor por la clave y la clave por el valor. En la figura 5.2.5.2.2 se muestra un ejemplo de como se podría realizar este tipo de consultas.

```
LPUSH list_Aguascalientes '1001' 'AGU' 'Aguascalientes'
LRANGE list_Aguascalientes 0 -1
```

Figura 5.2.5.2.2 Cambiando los valores a clave.

Regresando la subjerarquía de un nodo:

```
ZRANGEBYSCORE desc:32056 -inf +inf
```

Figura 5.2.5.2.3 Consulta de subjerarquía (Q4)

Hay que recordar que desc:32056 es un conjunto de todos los descendientes del municipio, y los dos últimos parámetros nos sirven para delimitar la profundidad de la subjerarquía deseada. En la figura 5.2.5.2.3 se muestran todos los descendientes con cualquier profundidad, si se requieren, por ejemplo, únicamente los descendientes directos, utilizaríamos el '1' en los dos últimos parámetros. En este caso se regresan todos los descendientes.

5.2.6 ZODB (Orientado a objetos)

La utilización y estandarización de las bases de datos orientadas a objetos se realizaron desde los años 80 [66]. Las tecnologías no se han propagado como muchos de los practicantes quisieran, sin embargo, el movimiento NoSQL ha sido un nuevo impulso para retomar algunas tecnologías y para crear otras. Dentro del modelo existen conceptos inherentes a las bases de datos, como la forma de persistencia, concurrencia, recuperación, consultas, y otros conceptos agregados a partir del orientado a objetos como encapsulación, tipos, herencia, instancias.

ZODB utiliza la estructura mostrada en la figura 5.2.6.1 en donde se define una forma de almacenamiento de acuerdo al lugar de persistencia y a la forma en que se guardan los objetos. Una vez realizada la conexión a la base de datos, se utiliza un contenedor, que es un diccionario de python, una vez que se agrega un objeto, se añade al almacenamiento físico y se está listo para guardar todas las modificaciones realizadas al objeto a través del tiempo. En la figura 5.2.6.2 se muestra un ejemplo de implementación de la estructura para iniciar la persistencia de los objetos

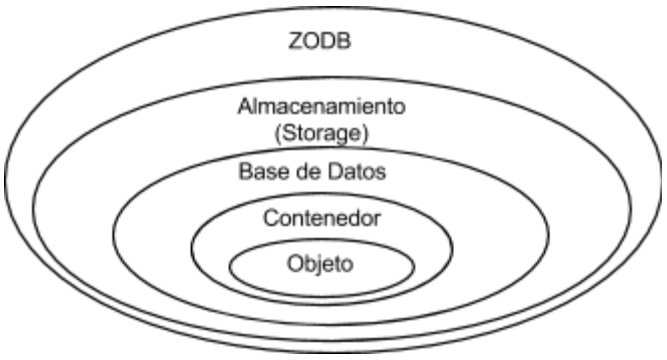


Figura 5.2.6.1 Modelo de ZODB

```
from ZODB import FileStorage, DB
storage = FileStorage.FileStorage('mydatabase.fs')
db = DB(storage)
connection = db.open()
root = connection.root()
```

Figura 5.2.6.2 Ejemplo de iniciación de almacenamiento en ZODB

Una vez creada la conexión, se agregó una instancia del contenedor, *root*, ahora es necesario crear clases que acepte la persistencia de ZODB para ello hay que heredar de la clase *Persistent* (Figura 5.2.6.3)

```
import ZODB
from Persistence import Persistent


class Libro(Persistent):
    def setTitulo(self, titulo):
        self.titulo = titulo
```

Figura 5.2.6.3 Clase persistente en ZODB

La clase Libro ahora es una subclase de *Persistent*, es posible agregarla a la base de datos (Figura 5.2.6.4) utilizando el contenedor de la base de datos. ZODB utiliza banderas para conocer si una instancia ha cambiado sus propiedades y así poder guardarla en la siguiente transacción.

```
libros=[]
for titulo in ['Cien años de soledad', 'Crónica de una muerte
anunciada', 'El coronel no tiene quien le escriba']:
    libro = Libro()
    libro.setTitulo(titulo)
    libros.append(libro)
root['libreria']=libros
transaction.commit()
```

Figura 5.2.6.4 Ejemplo de persistencia de objetos

5.2.6.1 Implementando la jerarquía

Existen tantos modelados como implementaciones de clases en una base de datos orientada a objetos. Los atributos de calidad son directamente expuestos en las implementaciones, el desempeño al recorrer un conjunto de datos, o la cantidad de datos regresados en una operación, pueden variar no sólo por la cantidad de información sino también por la forma en que son creadas las estructuras dentro de la aplicación.

```
#!/usr/bin/python
from treezodb import Tree
import ZODB, ZODB.FileStorage
import transaction

storage = ZODB.FileStorage.FileStorage('tl.fs')
db = ZODB.DB(storage)
connection = db.open()
root = connection.root
root.tree = Tree()
world = root.tree

world.create_node("Earth", "earth")
world.create_node("MX", "MX", parent="earth")
world.create_node("Aguascalientes", "AGU", parent="MX")
world.create_node("Aguascalientes", "1001", parent="AGU")
world.create_node("Aguascalientes", "Aguascalientes", parent="1001")
world.create_node("20000", "CP20000", parent="Aguascalientes")
world.create_node("Zona Centro", "P1", parent="CP20000")
world.create_node("Baja California", "BCN", parent="MX")
world.create_node("Mexicali", "2002", parent="BCN")
world.create_node("Mexicali", "Mexicali", parent="2002")
world.create_node("21399", "CP21399", parent="Mexicali")
world.create_node("Los Laureles", "P1782", parent="CP21399")
world.create_node("Zacatecas", "ZAC", parent="MX")
world.create_node("Zacatecas", "32056", parent="ZAC")
world.create_node("Zacatecas", "Zacatecas", parent="32056")
world.create_node("98000", "CP98000", parent="Zacatecas")
world.create_node("Zacatecas Centro", "P75004", parent="CP98000")
transaction.commit()
```

Figura 5.2.6.1.1 Implementación en Python y persistencia en ZODB

Para la investigación se utilizó una estructura de árbol desarrollada por Brett Kromkamp, Hsiamin Chen, Ilya Kuprik y Holger Bast [67], elegida por las implementaciones de las operaciones inherentes a los árboles, desde la creación de nodos, hasta el despliegue y desarrollo de un subárbol. Así como en la figura 5.2.6.3 se creó una subclase a partir de Persistent, como se muestra en la figura 5.2.6.1.1 se utiliza la clase Tree del módulo treezodb, esta clase es una modificación de la implementación del árbol [68] para aceptar la persistencia en ZODB y alertar los cambios en la estructura para que puedan ser guardados.

Estrategia de modelado elegida	Árbol (pyTree)
Justificación	La jerarquía utilizada es un árbol, las operaciones necesarias y las búsquedas se encuentran implementadas, fácil implementación.
Otras alternativas	Grafo (python-graph), Otros árboles (ETE Treenode)

Figura 5.2.6.1.2 Estrategias de modelado en ZODB

5.2.6.2 Consultando la jerarquía

En la implementación de árbol utilizada [67] ya se cuentan con funciones de búsqueda, creación, borrado y obtención de subjerarquías.

Regresando un nodo por su identificador:

```
def get_node(self, nid):
        if nid is None or not self.contains(nid):
            return None
        return self._nodes[nid]

tree.get_node("CP01080")
```

Figura 5.2.6.2.1 Consulta por id (Q2)

Se define una función para encontrar el id, si no se encuentra se regresa None. La línea de código de abajo de la función de la figura 5.2.6.2.1 muestra el llamado durante la implementación del árbol.

Regresando los nodos por el valor de un atributo:

En la figura 5.2.6.2.2 se muestra una búsqueda lineal sobre la propiedad del nodo tag, que fue utilizado como nombre de los nodos. Esta búsqueda puede ser mejorada a nivel de desempeño, al agregar búsquedas no lineales y multiprocesamiento en la implementación.

```
def search_tag(self, tag):
     nodes = []
     for nodeid in self.expand_tree():
         if(self[nodeid].tag == tag):
          nodes.append(self[nodeid])
     return nodes
```

Figura 5.2.6.2.2 Cambiando los valores a clave. (Q3)

Regresando la subjerarquía de un nodo:

```
def subtree(self, nid):
        st = Tree()
        if nid is None:
            return st

        if not self.contains(nid):
            raise NodeIDAbsentError("Node '%s' is not in the
tree" % nid)

        st.root = nid
        for node_n in self.expand_tree(nid):
            st._nodes.update({self[node_n].identifier : self
[node_n]})
        return st
```

Figura 5.2.6.2.3 Consulta de subjerarquía (Q4)

La función implementada genera un árbol nuevo con el nodo enviado en los parámetros como nodo raíz.

5.2.7 Replicación del experimento

Para realizar una réplica del experimento, o realizar pruebas sobre las diferentes tecnologías y modelados se realizaron ambientes de desarrollo sobre vagrant. Todas las máquinas se realizaron con la versión 1.2.2 y corriendo una máquina virtual con Ubuntu 12.04 LTS. Para la instalación de vagrant y correr una máquina virtual específica se puede consultar el Apéndice 3.

5.3 Experimentación

La comparación se realizó sobre el desempeño de cada base de datos al ejecutar las cuatro consultas implementadas en la sección anterior (inserción de datos, consulta por id, consulta por atributo y subjerarquía). Se midió a partir del tiempo de memoria y CPU utilizando los valores de salida *user* y *sys* del comando TIME [69] (Figura 5.3.1). El experimento se realizó sobre las seis máquinas de vagrant independientemente y de manera alterna.

Para conocer el comportamiento de las bases de datos ante el crecimiento de la información, se estableció aumentar la cantidad de nodos 20% hasta llegar al total de nodos de 107,630 . En la tabla 5.3.1 se muestran los porcentajes con el número de líneas leídas y su correspondiente en nodos.

Porcentaje de información	Líneas leídas	Nodos creados
20%	15041	20941
40%	30081	42497
60%	45122	64089
80%	60162	86374
100%	75203	107630

Tabla 5.3.1 Nodos implementados de la jerarquía

```
time nice -0 ./q2-id.py
```

Figura 5.3.1 Ejemplo de ejecución de una consulta

En la figura 5.3.2 se muestra un resumen de las cuatro consultas comparadas, es importante recordar la ausencia de Redis en la consulta por valor de atributo de acuerdo al paradigma de clave-valor. Los valores mostrados son los tiempos totales de CPU (*user* + *sys*) para cada consulta realizada por cada una de las tecnologías.

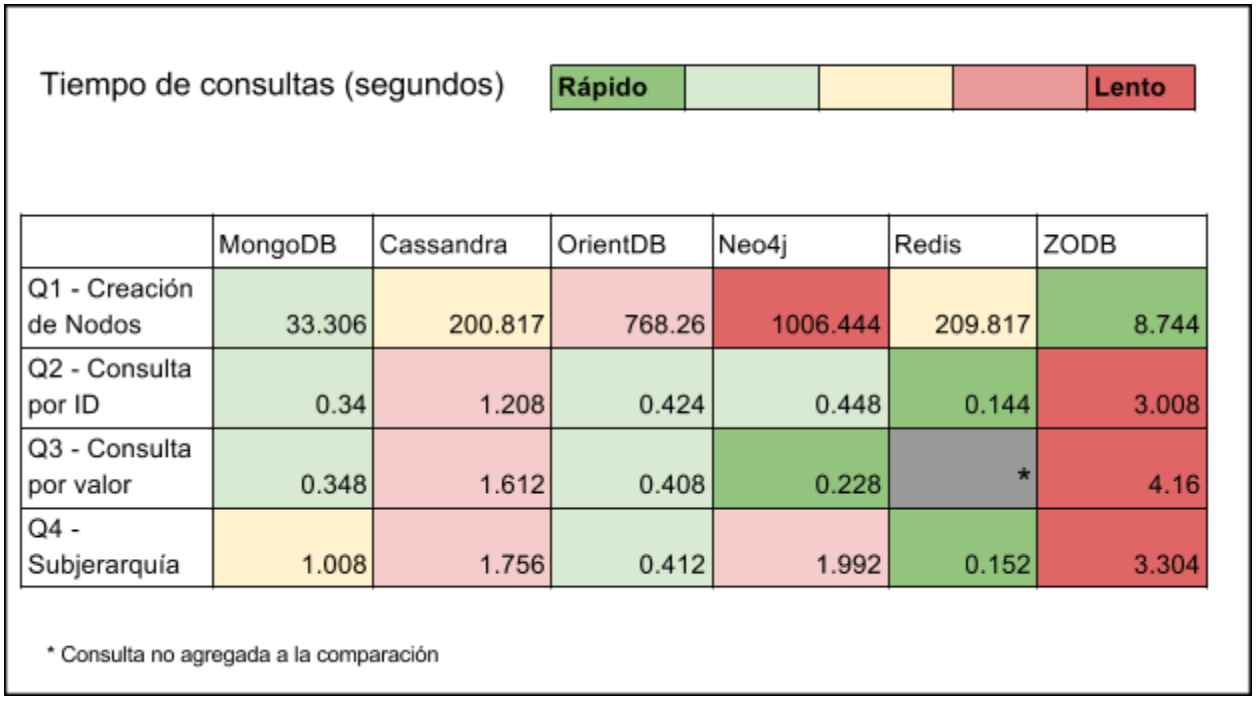


Figura 5.3.2 Resumen de tiempo de CPU de las consultas

5.3.1 Q1 - Creación de nodos

Dentro de las consultas comparadas esta es la única de escritura. Las consultas de escritura consumen mayores recursos. Para el caso a esta primera operación se utilizó una escala mayor para una mejor visualización de los datos. El proceso se realizó a partir de la lectura de los datos [48], generando un total de 107,630 nodos de acuerdo a la estructura definida. En la tabla 5.3.1.1 y en la figura 5.3.1.1 se muestran los resultados y el comportamiento de cada tecnología al incremento de la información ($\Delta = 20\%$).

La base de datos más rápida fue la orientada a objetos, agregando a la implementación del árbol y haciéndola persistente en un tiempo menor a nueve segundos. Las bases de datos orientadas que soportan grafos, son las de menor desempeño, ya que utilizan mayor cantidad de operaciones para generar y mantener el grafo.

	20%	40%	60%	80%	100%
MongoDB	7.148	13.701	20.315	26.974	33.306
Cassandra	41.427	79.437	118.372	163.586	200.817
OrientDB	152.029	304.863	454.696	613.551	768.26
Neo4j	161.386	357.334	496.639	734.638	1006.444
Redis	41.222	83.261	123.179	168.066	209.817
ZODB	2.224	3.736	5.244	6.848	8.744

Tabla 5.3.1.1 Resultados del tiempo de CPU en segundos conforme al incremento de información

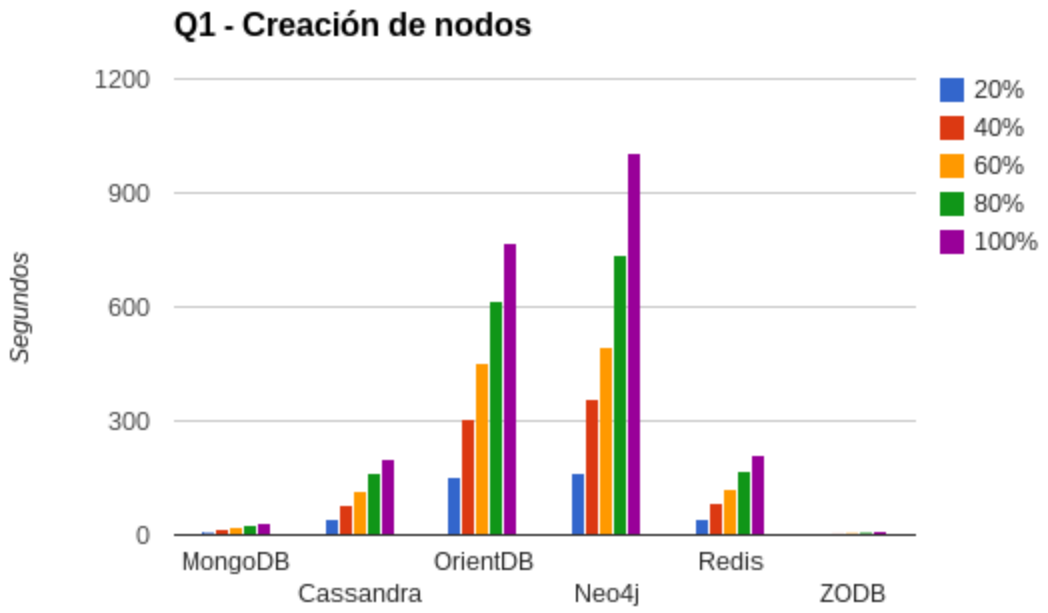


Figura 5.3.1.1 Tiempo de CPU en segundos para la creación de nodos

En la figura 5.3.1.1 se muestra el incremento del tiempo en todas las bases de datos conforme se aumenta la cantidad de información, no así el throughput (cantidad de nodos por segundo), como se muestra en la figura 5.3.1.2, en la cual se mantiene constante. Se obtuvo un máximo por parte de ZODB de 12,613 nodos insertados por segundo y un mínimo por parte de Neo4j de 107 nodos/seg.

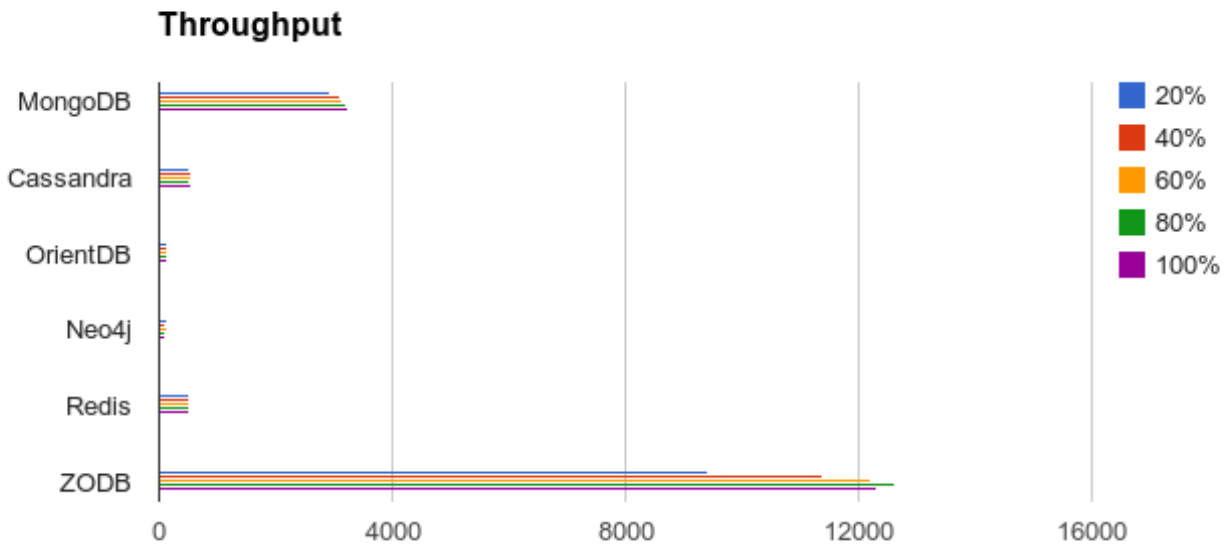


Figura 5.3.1.2 Throughput de la consulta de creación de nodos

5.3.2 Q2 - Consulta de nodo por identificador

Las consultas por *id* pueden variar dependiendo de la base de datos, en Neo4j y OrientDB por ejemplo, se les agregan automáticamente los identificadores a cada nodo, en MongoDB y Cassandra se pueden agregar distintos identificadores que se agregan como índices y finalmente en Redis y la implementación utilizada en ZODB es obligatorio el agregar un *id* o llave. Los resultados mostrados son conforme a la búsqueda por identificador descrita en la sección anterior, obteniendo un sólo nodo de toda la jerarquía.

	20%	40%	60%	80%	100%
MongoDB	0.344	0.352	0.352	0.356	0.34
Cassandra	1.016	1.14	1.148	1.076	1.208
OrientDB	0.388	0.392	0.424	0.404	0.424
Neo4j	0.28	0.628	0.456	0.636	0.448
Redis	0.132	0.136	0.128	0.136	0.144
ZODB	0.88	1.336	1.944	2.248	3.008

Tabla 5.3.2.1 Resultados de los tiempos de CPU de la consulta por id en segundos

Redis fue diseñada para encontrar valores a partir de una llave, por lo que es la más rápida con un poco más de 100 milisegundos (Tabla 5.3.2.1). ZODB es la única base de datos, que por la implementación refleja cambios de desempeño cuando una mayor cantidad de información es ingresada a la jerarquía (Figura 5.3.2.1)

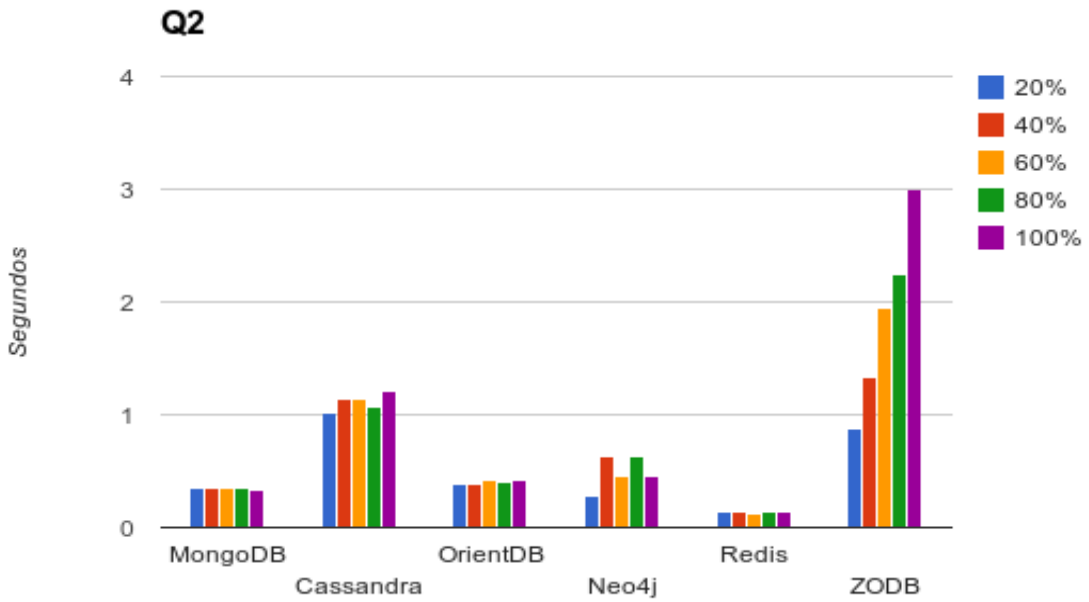


Figura 5.3.2.1 Tiempos de CPU de la consulta por id en segundos.

5.3.3 Q3 - Consulta de nodo por valor

Recordando que se utilizaron las configuraciones por defecto, es posible cambiar los resultados encontrados en esta consulta con diversas estrategias, una de ellas es la creación de índices, además en tecnologías como Cassandra, se puede utilizar la partición de la información, por lo que sería posible clusterizar la información y mejorar el rendimiento de este tipo de consultas. En el caso de la implementación en ZODB, se realizó una búsqueda lineal, siendo ésta la más básica, a través de todo el árbol, por lo que la implementación de diferentes algoritmos de búsqueda también cambiarían sus resultados.

	20%	40%	60%	80%	100%
MongoDB	0.348	0.34	0.312	0.332	0.348
Cassandra	1.16	1.22	1.3	1.52	1.612
OrientDB	0.396	0.396	0.416	0.412	0.408
Neo4j	0.264	0.28	0.26	0.304	0.228
ZODB	1.028	1.792	2.576	3.232	4.16

Tabla 5.3.3.1 Resultados de los tiempos de CPU de la consulta por valor

Como se muestra en la tabla 5.3.3.1 y en la figura 5.3.3.1 se excluyó Redis de esta comparación, aunque es posible simular la consulta al cambiar los valores como claves y las claves como valores, no se realizó para respetar la naturaleza de la base de datos y evaluar exclusivamente las consultas por valor, sin embargo, es posible que en implementaciones externas se vean listas y conjuntos que utilicen los valores como claves.

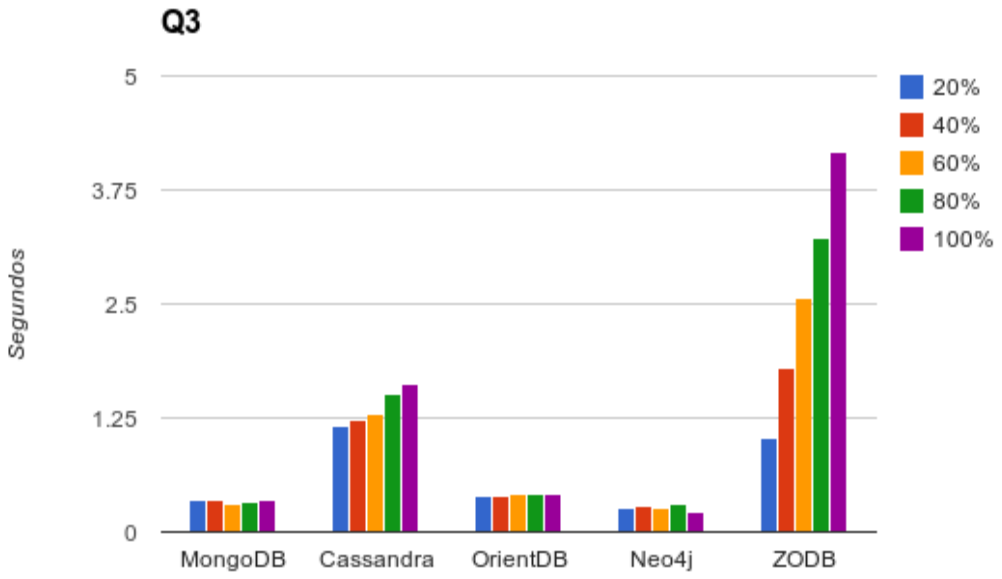


Figura 5.3.3.1 Tiempos de CPU de la consulta por valor en segundos.

5.3.4 Q4 - Subjerarquía

La consulta de todos los descendientes de un nodo es de las más comunes dentro de una jerarquía. A diferencia de las consultas por id y por valor, la tabla 5.3.4.1 refleja el cambio de desempeño, para las bases de datos que no soportan grafos, durante el incremento de la información ante una misma consulta de toda la subjerarquía de un nodo.

	20%	40%	60%	80%	100%
MongoDB	0.596	0.912	0.896	1.072	1.008
Cassandra	1.544	1.576	1.62	1.756	1.756
OrientDB	0.436	0.416	0.412	0.416	0.412
Neo4j	2.12	2.208	1.704	1.652	1.992
Redis	0.14	0.128	0.132	0.148	0.152
ZODB	1.056	1.508	1.984	2.468	3.304

Tabla 5.3.4.1 Resultados de los tiempos de CPU de la consulta de subjerarquía

A pesar del gran desempeño mostrado por Redis (Figura 5.3.4.1), hay que mencionar que la implementación mostrada en la sección 5.2.5 fue diseñada para realizar este tipo de consultas al incorporar conjuntos con puntaje, para simular la profundidad de la jerarquía. En bases de datos que ofrecen una gran flexibilidad, como Redis, es recomendable pensar en los tipos de consulta que se vayan a realizar, esto puede impactar de manera positiva en el rendimiento de las soluciones.

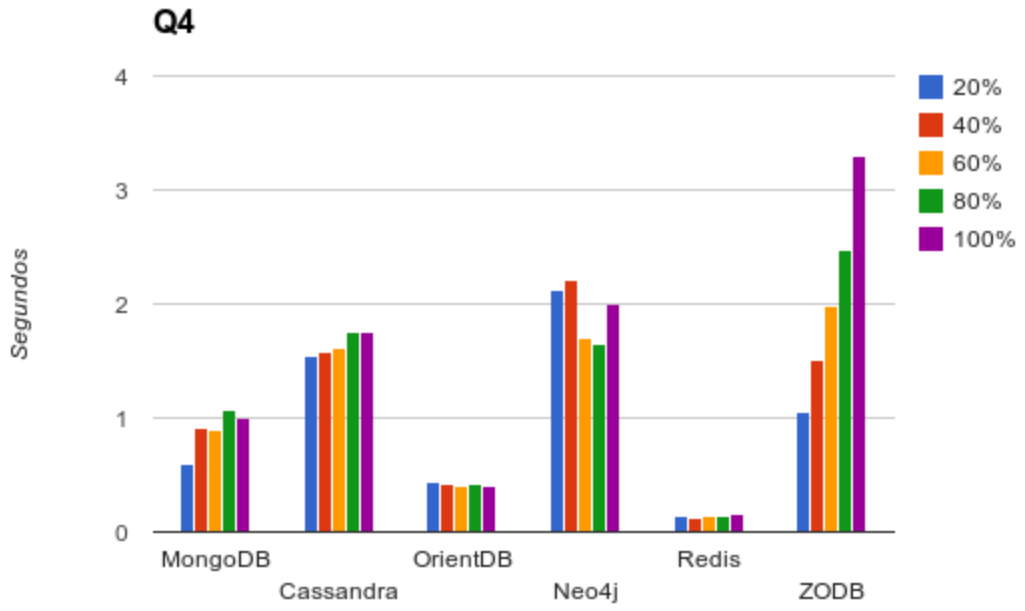


Figura 5.3.4.1 Tiempos de CPU de la consulta de jerarquía en segundos.

Si se desea ver todos los resultados obtenidos a lo largo de la investigación, además de gráficas comparativas y promedios, se puede consultar la hoja de cálculos (http://goo.gl/7sgB8d)

5.3.5 Dependencia a Entradas/Salidas ó a CPU

Uno de los hallazgos dentro de la investigación y mostrados en la tabla 5.3.5.1, fue el determinar si cada una de las consultas estaban limitadas por las escrituras/lecturas a disco (*I/O bound*) o por el procesamiento (*CPU bound*). Para esta investigación se considera I/O bound cuando el tiempo real sea más del doble al tiempo de CPU (user + sys), de lo contrario se considera CPU bound.

El conocimiento de las consultas de una tecnología puede apoyar además en la selección de Hardware para alojar el servicio de base de datos. Para aquellos procesos que se determinen como I/O bound se pueden utilizar discos duros más rápidos para mejorar el desempeño y, para aquellos que son CPU bound, la utilización de multiprocesamiento y procesadores más eficientes afectan directamente los tiempos de la consulta.

	MongoDB	Cassandra	OrientDB	Neo4j	Redis	ZODB
Q1 - Creación de nodos	I/O Bound	I/O Bound	I/O Bound	I/O Bound	I/O Bound	CPU Bound
Q2- Consulta por ID	CPU Bound	I/O Bound	CPU Bound	CPU Bound	CPU Bound	CPU Bound
Q3 - Consulta por valor	I/O Bound	I/O Bound	I/O Bound	I/O Bound	*	CPU Bound
Q4 - Subjerarquía	I/O Bound	CPU Bound	I/O Bound	CPU Bound	CPU Bound	CPU Bound

* Consulta no agregada a la comparación

Tabla 5.3.5.1 Resumen de consultas dependientes a entradas y salidas o a CPU

6 Conclusiones

A continuación se presentan el alcance logrado respecto a los objetivos establecidos para la presente investigación y además las área de oportunidad y hallazgos de la presente tesis.

Objetivo 1. Mostrar la implementación de cada tecnología NoSQL ante un problema específico de datos jerárquicos, evidenciando la forma en que la solución se ajusta al problema.

Para este objetivo en particular, las implementaciones realizadas hacen evidente ciertas características que son buscadas por desarrolladores de soluciones bajo diferentes circunstancias.

- La **flexibilidad** inherente de algunas tecnologías como MongoDB y Redis las convierte en herramientas útiles de propósito general.
- La mayor **naturalidad** en la solución del problema propuesto en las bases de datos que aceptan grafos como lo son OrientDB y Neo4j. Este tipo de base de datos cuentan con operadores, consultas y herramientas que facilitan la visualización y manejo de los datos expuestos, además de diferenciar aspectos más específicos como los componentes de la jerarquía, los nodos y sus relaciones.
- La **claridad reflejada en la simplicidad** que dan las bases de datos orientadas a objetos. ZODB elimina las diferencias de paradigmas entre el código y la base de datos.
- Por último, Cassandra y OrientDB ofrecen, por medio de sus lenguajes de consulta, **similitud** a SQL, lenguaje ampliamente aceptado, lo que ayuda durante la adaptación de las nuevas tecnologías.

El observar las implementaciones ayuda a tener un panorama general de la capacidad para solucionar problemas específicos para cada tecnología y modelado utilizado. Cada implementación contribuye a favor o en detrimento de algún atributo deseado. Por ejemplo, a veces es necesario un esquema fijo para evitar inconsistencias entre registros. Ello da pie para evaluar comparaciones existentes entre modelos o tecnologías para examinar características más puntuales o problemas diferentes a la administración de datos jerárquicos (Capítulo 2).

Objetivo 2. Replicar el experimento diseñado con una jerarquía específica, para este caso la composición geográfica a diferencia de datos heterogéneos[8], para contrastar las conclusiones del experimento y su réplica.

Respecto a este objetivo se corroboraron algunas de las discusiones finales de dicha investigación:

- Las bases de datos orientadas a documentos (MongoDB), tuvieron el mejor desempeño general a lo largo de las diferentes consultas.
- Las bases de datos orientadas a grafos (OrientDB y Neo4j), destacaron durante la recuperación de datos, sin embargo son las que necesitan mayores recursos durante la inserción de nodos.
- La facilidad de implementación y entendimiento es una ventaja para las bases de datos orientadas a grafos.

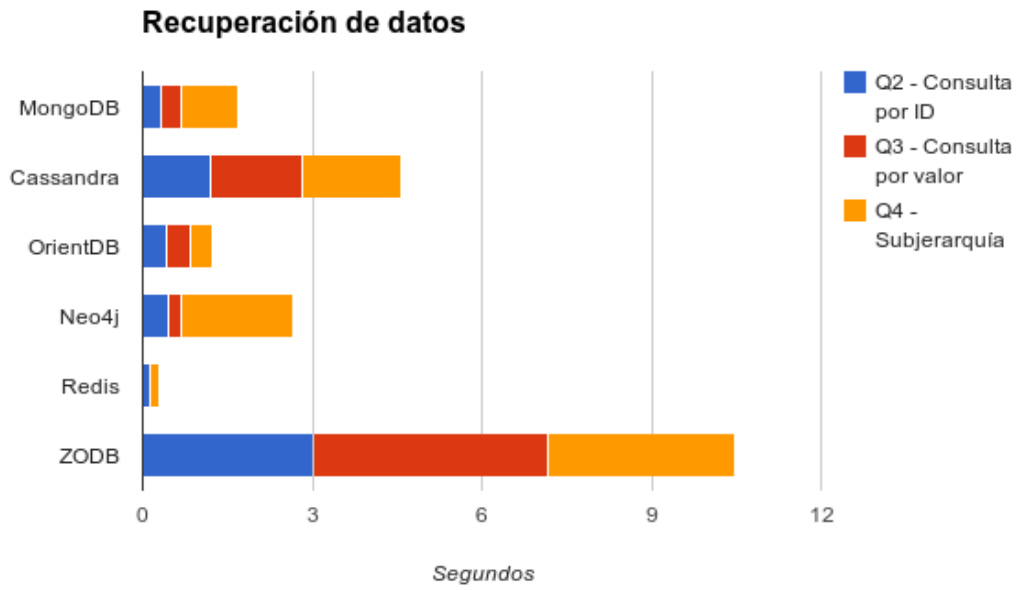


Figura 6.1 Gráfica de las consultas de recuperación de datos

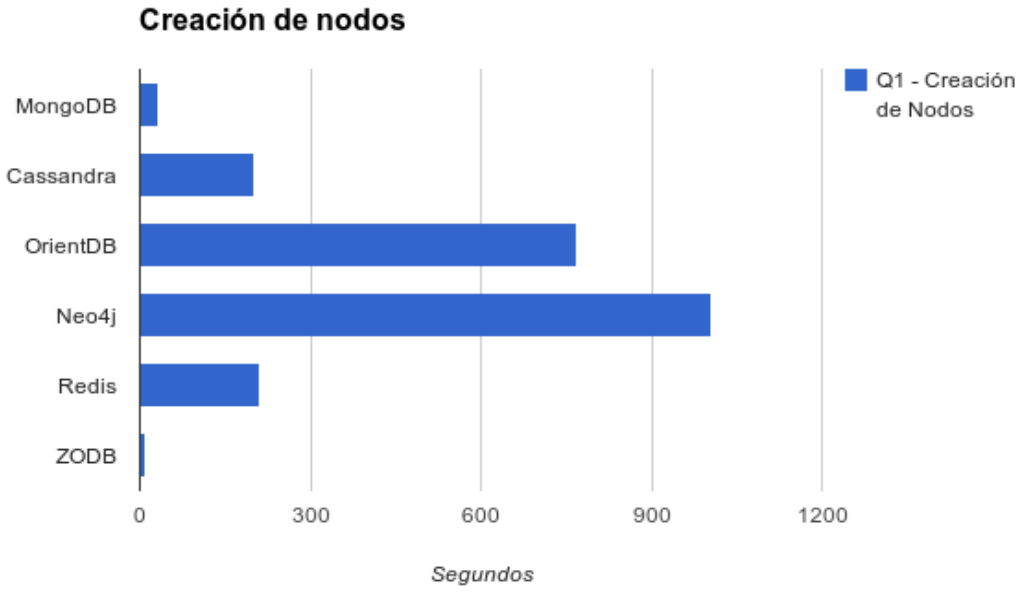


Figura 6.2 Comparación en segundos sobre la creación de nodos

Objetivo 3. Extender el experimento base, agregando la base de datos orientada a objetos a las comparaciones y conclusiones. Las figuras 6.1 y 6.2 ayudan a visualizar estas hallazgos, también hay que añadir a las conclusiones:

- Las bases de datos orientadas a objetos (ZODB) obtuvieron los mejores tiempos para la inserción de datos, pero el peor desempeño al recuperar información con la estructura utilizada y los algoritmos de búsqueda implementados.
- Las bases de datos orientadas a objetos (ZODB) fueron las más susceptibles a los cambios realizados en la cantidad de información, y OrientDB (Multimodelo) y Redis (Clave-Valor) las más estables durante el incremento de nodos.

Objetivo 4. Crear las máquinas virtuales de cada una de las tecnologías seleccionadas para su fácil replicación y que sirvan como base para experimentos posteriores.

El objetivo fue realizado de una manera sencilla y eficiente gracias a Vagrant [46]. Todas las máquinas virtuales cuentan con el experimento cargado para una fácil ejecución. Se puede consultar la instalación para la réplica del experimento en el apéndice III.

6.1 Reflexiones y Hallazgos

Una consulta puede estar limitada por lecturas/escrituras a disco (I/O bound) o al procesamiento (CPU bound). El conocer este dato puede ayudar a optimizar el desempeño de la tecnología y modelo utilizado a través de la inversión en el recurso adecuado.

La implementación de la tecnología y modelo para cada tipo de base de datos fue llevada a cabo en un tiempo fijo, evitando así, ajustes detallados y configuraciones a medida para la mejora del desempeño. Por lo tanto, otra área de oportunidad es la optimización de los motores de base de datos para la aplicación del mismo experimento.

La propia evolución de cada tecnología hace natural que replicar este experimento pueda traer resultados diferentes. Ejemplo visto durante la investigación es la evolución del modelado de Cassandra y el cambio a multimodelo de Membase (Couchbase). Ello puede traer resultados diferentes aún realizando en un futuro el mismo experimento.

La intención de este tipo de estudios es ayudar a la selección de una tecnología para un problema específico y es útil por la gran cantidad de alternativas existentes, para el caso más de 150 ejemplos de bases de datos NoSQL [5].

6.2 Trabajo futuro

El experimento fue diseñado para en un futuro poder escalarlo con toda la información que se encuentra en Geonames [47], agregando la información de todos los países que tengan una misma estructura geográfica. También es posible escalar la cantidad de información y propiedades de cada nodo, para así, no sólo ver el comportamiento de la tecnología en la creación de nodos, sino también durante la recuperación de datos y la flexibilidad del modelado al incorporar nueva información.

La comparación se realizó utilizando un sólo modelo de datos para cada tecnología, con la extensión de diferentes modelados de jerarquías implementadas en la misma tecnología, sería posible ver diferentes alternativas de representaciones de jerarquías y el desempeño logrado por cada una de las estrategias más características.

Además de las consultas realizadas en la investigación: creación de nodos, consulta por id, consulta por valor y subjerarquía, existen otras operaciones características de los datos jerárquicos como agregar el borrado y traslado de nodos o consultas entre niveles de la jerarquía, las cuales también pudieran ser agregadas o consideradas en futuros experimentos.

La investigación incorporó una tecnología por cada modelo NoSQL. El incorporar más tecnologías podría servir para observar las características principales de cada base de datos y cómo su presencia o ausencia afecta el desempeño.

Además, durante el estudio de las comparaciones en el ambiente NoSQL, se observó la falta de estudios profundos entre un mismo modelo, una oportunidad clara fue identificada en el modelo orientado a documentos, el cual ha logrado una fuerte aceptación en años recientes.

Existen diversas tecnologías NoSQL con características que solventan mejor problemas específicos. Un área de oportunidad en este sentido es analizar un problema específico con solo un tipo de modelo o tecnología NoSQL.

Los resultados que concluyen la relación entre las diferentes consultas y el tiempo real del proceso (Tabla 5.3.5.1), generando una posibilidad de una dependencia de Entradas/Salidas (I/O Bound) o del procesador (CPU Bound), pueden ser verificadas a detalle por medio de un análisis profundo con herramientas como iotop y vmstat [69].

La arquitectura de una aplicación está fuertemente relacionada con la forma en que se guardan y se consultan los datos, es por eso que la elección razonada de una base de datos (o varias) es importante.

A través de esta investigación, se espera que el lector haya identificado algunas ventajas y limitaciones relevantes de diversos tipos de modelados en el ambiente NoSQL y pueda reconocer alguno como alternativa de solución. A partir de este momento, sólo queda el interés que se haya generado por analizar a detalle alguna tecnología u otras opciones que existen allá afuera para persistir los datos.

Apéndice I

Teorema CAP

Eric Brewer en su plática “Towards robust distributed systems” [70], y la demostración formal realizada por Seth Gilbert y Nancy Lynch [71], menciona tres características en servicios web, la consistencia, disponibilidad y particiones del servicio. A pesar de que se desean las tres características, la inclinación hacia alguna característica puede comprometer una o las dos características restantes, por lo que se tienen que seleccionar como máximo dos de los atributos (Figura AI.1).

Figura AI.1 Teorema CAP

Para ver instancias de implementaciones particulares con tecnologías representantes se puede consultar la figura 4.1

ACID vs BASE

Brewer también aborda el contraste entre la consistencia y la disponibilidad, y lo demuestra con las transacciones ACID vs BASE (Figura AI.2)

Figura AI.2 ACID vs BASE

El movimiento NoSQL tiene mayor presencia en las características AP y CP (Figura AI.1), ya que la parte de consistencia y disponibilidad (CA), está mayormente cubierta por las bases de datos relacionales. Además, la mayoría de las alternativas NoSQL incorporan el manejo de *Big Data*, lo que es necesario incorporar particiones para lograr una escalabilidad de la solución, algunas de ellas buscando la consistencia en la información, utilizando transacciones ACID y otras la disponibilidad del servicio, al realizar transacciones BASE (Figura AI.2).

Apéndice II

Tabla de características generales

La tabla AII.1 contiene información acerca de las características generales de los seis manejadores de bases de datos que se estudiaron dentro de la investigación (MongoDB, Cassandra, OrientDB, Neo4j, Redis, ZODB).

	MongoDB	Cassandra	OrientDB	Neo4j	Redis	ZODB
Modelo	Documentos	Columnas	Documentos /Grafos	Grafos	Clave-Valor	Objetos
Licencia	AGPL	Apache 2.0	Apache 2.0	GPL	BSD	ZPL
Versión	2.6	2.0.8	1.7.2	2.0.3	2.8.10	4.0
Primer liberado	2009	2008	2010	2007	2009	2005
Tipos de datos	JSON (String, Array, Object)	String, integer, blob, boolean, float, uuid	String, byte, float, date, integer, link, custom	String, byte, float, date, integer, char	Set String Lists Hash	Objetos en Python
Relaciones	N/A	N/A	Aristas	Aristas	N/A	Atributo
Formato del registro	JSON/ BSON	Tabla	JSON	Hash	String	Objeto
Lenguaje	C++	Java	Java	Java	C	Python
REST	Sí	No	Sí	Sí	No	No
Consulta	Comandos/ Javascript	CQL	SQL/Gremlin	Cypher/ Gremlin	Comandos	Python
Mapreduce	Javascript	Hadoop	N/A	N/A	N/A	N/A
Replicación	Maestro - Esclavo	Multi-Maestro	Multi-Maestro	Maestro - Esclavo	Maestro - Esclavo	Maestro - Esclavo
Particiones	Sí	Sí	Sí	No	Sí	No
Concurrencia	Bloqueo en escritura	Timestamp	Bloqueo en escritura	Bloqueo en escritura	N/A	Bloqueo en escritura
Transacciones	Cambios en dos fases (temporal - durable)	BASE - ACID	ACID	ACID	Comandos en cola	ACID
Seguridad	Usuarios	Usuarios	Usuarios	N/A	Password	N/A
Teorema CAP	CP	AP	AP	CA	CP	CA

Tabla AII.1 Características generales de los manejadores de bases de datos

Apéndice III

Instalación de Vagrant para la replicación del experimento

Esta es una guía para la instalación de Vagrant sobre Ubuntu 12.04 y la implementación de las 6 máquinas virtuales con las bases de datos de esta investigación.

AIII.1 Instalación de VirtualBox

Vagrant corre sobre algún ambiente de máquinas virtuales [73], por defecto utiliza VirtualBox[74], aunque cuenta con soporte para VMWare.

Este paso es opcional, al momento de instalar Vagrant (AIII.2) se busca y se instala VirtualBox automáticamente, si se requiere instalar de forma manual puede seguir los siguientes pasos:

1) Agregar a la lista de paquetes

deb http://download.virtualbox.org/virtualbox/debian precise contrib a ***/etc/apt/sources.list***

2) Descargar y registrar la llave

wget -q http://download.virtualbox.org/virtualbox/debian/oracle_vbox.asc -O- | sudo apt-key add -

3) Actualizar e instalar VirtualBox

sudo apt-get update

sudo apt-get install virtualbox-4.3

AIII.2 Instalación de Vagrant

Instalación por medio de apt

1) Instalar Vagrant

sudo apt-get install vagrant

Es necesaria la versión 1.2.2 o posterior también encontrada en la página de descargas de vagrant (http://downloads.vagrantup.com/)

AIII.3 Instalación de la máquina virtual

A continuación los pasos para instalar neo4j. Se realizan los mismos pasos para cualquier otra base de datos.

1) Para instalar una máquina virtual se recomienda agregar una nueva carpeta para cada tecnología:

mkdir $HOME/neo4j

cd $HOME/neo4j

2) Descargar la máquina con la base de datos (utilizar las ligas de la tablaAIII.3.1)

wget http://rumayor.org/thesis/experiment/boxes/neo4j/neo4j.box

3) Configuración de la máquina de vagrant recién descargada

vagrant init neo4j neo4j.box

4) Correr la máquina virtual

vagrant up

5) Ingresar a la máquina virtual

vagrant ssh

Base de Datos	Liga a máquina virtual
MongoDB	http://rumayor.org/thesis/experiment/boxes/mongodb/mongodb.box
Cassandra	http://rumayor.org/thesis/experiment/boxes/cassandra/cassandra.box
OrientDB	http://rumayor.org/thesis/experiment/boxes/orientdb/orientdb.box
Neo4j	http://rumayor.org/thesis/experiment/boxes/neo4j/neo4j.box
Redis	http://rumayor.org/thesis/experiment/boxes/redis/redis.box
ZODB	http://rumayor.org/thesis/experiment/boxes/zodb/zodb.box

Tabla AIII.3.1 Ligas a las máquinas virtuales

Para más información y detalles sobre el manejo de vagrant se recomienda seguir la documentación oficial [46].

Las diversas máquinas virtuales ya cuentan con el experimento cargado, si se requiere mayor información para el manejo de la información del experimento se puede consultar el repositorio del experimento (https://github.com/AgusRumayor/nosql)

Referencias

[1] Codd, E. F. (1970). A relational model of data for large shared data banks.Communications of the ACM, 13(6), 377-387.

[2] Leavitt, N. (2010). Will NoSQL databases live up to their promise?. Computer,43(2), 12-14.

[3] Strozzi, C. (2011). NoSQL-A Relational Database Management System. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL. Accesado el 7 de Febrero del 2014

[4] Tudorica, B. G., & Bucur, C. (2011, Junio). A comparison between several NoSQL databases with comments and notes. In Roedunet International Conference (RoEduNet), 2011 10th (pp. 1-5). IEEE.

[5] Edlich, Stefan, "NoSQL, your ultimate guide to the non - relational universe!", http://nosql-database.org/, (Sin publicar). Accesado el 15 de Febrero del 2014.

[6] DB-Engines. DB-Engines Encyclopedia. http://db-engines.com/en/articles. Accesado el 18 de Marzo del 2014

[7] Celko, J. (2012). Joe Celko's Trees and hierarchies in SQL for smarties.

[8] Jayathilake, D., Sooriaarachchi, C., Gunawardena, T., Kulasuriya, B., & Dayaratne, T. (2012, September). A study into the capabilities of NoSQL databases in handling a highly heterogeneous tree. In Information and Automation for Sustainability (ICIAfS), 2012 IEEE 6th International Conference on (pp. 106-111). IEEE.

[9] Brooks Jr, F. (1986). NO SILVER BULLET ESSENCE AND ACCIDENTS OF SOFTWAR ENGINEERING.

[10] MongoDB. Model Tree Structures in MongoDB. http://docs.mongodb.org/manual/tutorial/model-tree-structures/. Accesado el 20 de Febrero del 2014

[11] Faure-Lacroix L. (2013, Julio). Storing Hierarchical Data in CouchDB. http://blog.vosnax.ru/2013/07/18/Storing-Hierarchical-Data-in-CouchDB/. Accesado el 11 de Marzo del 2014

[12] Google groups: Redis DB. (2010, Julio) Hierarchical trees. https://groups.google.com/forum/#!topic/redis-db/IsLJ4PIBo9E. Accesado el 11 de Marzo del 2014

[13] Pokorny, J. (2013). NoSQL databases: a step to database scalability in web environment. International Journal of Web Information Systems, 9(1), 69-82.

[14] RethinkDB. RethinkDB compared to MongoDB. http://www.rethinkdb.com/docs/rethinkdb-vs-mongodb/. Accesado el 20 de Marzo del 2014.

[15] Abramova, V., & Bernardino, J. (2013, Julio). NoSQL databases: MongoDB vs cassandra. In Proceedings of the International C* Conference on Computer Science and Software Engineering (pp. 14-22). ACM.

[16] DB-Engines. DB-Engines Ranking. http://db-engines.com/en/ranking. Accesado el 18 de Marzo del 2014.

[17] VSChart.com. VSChart.com The comparison wiki. http://vschart.com/. Accesado el 3 de Marzo de 2014.

[18] FindTheBest. Best NoSQL Databases - Features & Specs. http://nosql.findthebest.com/. Accesado e 20 de Marzo del 2014.

[19] Badrit. Redis vs. MongoDB Performance. (2013, Noviembre). http://www.badrit.com/blog/2013/11/18/redis-vs-mongodb-performance. Accesado el 23 de Marzo del 2014.

[20] Narde, Rohan. A Comparison of NoSQL systems. Diss. Rochester Institute of Technology, 2013.

[21] Hecht, R. (2011). NoSQL Evaluation: A use case oriented survey, 2011 International Conference on. IEEE, 2011, pp. 336-341.

[22] Han, J., Haihong, E., Le, G., & Du, J. (2011, October). Survey on NoSQL database. In Pervasive computing and applications (ICPCA), 2011 6th international conference on (pp. 363-366). IEEE.

[23] Cronin, Devlin. (2012) "A Survey of Modern Key-Value Stores." . En Cal Poly Computer Science Department Labs.

[24] Ortíz, Yudisney Vazquez, and Anthony Rafael Sotolongo León. (2013) "Mirada a bases de datos NoSQL de código abierto orientadas a documentos." Serie Científica 6.9 .

[25] Tinkerpop. Blueprints. https://github.com/tinkerpop/blueprints/wiki. Accesado el 24 de Marzo del 2014.

[26] Pasarin Perea, L. (2013). Analysis of alternatives to store genealogical trees using Graph Databases.

[27] Ciglan, M., Averbuch, A., & Hluchy, L. (2012, Abril). Benchmarking traversal operations over graph databases. In Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on (pp. 186-189). IEEE.

[28] Bartholomew, D. (2010). SQL vs. NoSQL. Linux Journal, 2010(195), 4.

[29] Wei-ping, Z., Ming-Xin, L., & Huan, C. (2011, Mayo). Using MongoDB to implement textbook management system instead of MySQL. InCommunication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on (pp. 303-305). IEEE.

[30] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010, April). A comparison of graph database and a relational database: a data provenance perspective. In Proceedings of the 48th annual Southeast regional conference (p. 42). ACM.

[31] Hadjigeorgiou, C. (2013). RDBMS vs NoSQL: Performance and Scaling Comparison. The University of Edinburgh

[32] Datastax. (2013). About Data Consistency in Cassandra. http://www.datastax.com/docs/0.8/dml/data_consistency. Accesado el 24 de Marzo del 2014.

[33] MongoDB. (2013). Introduction to MongoDB. http://www.mongodb.org/about/introduction/. Accesado el 24 de Marzo del 2014.

[34] Celko, J. (2012). Joe Celko's Trees and hierarchies in SQL for smarties. Elsevier.

[35] Feasel, J. (2012). SQL Fiddle. http://sqlfiddle.com/. Accesado el 24 de Marzo del 2014

[36] Redmond, E., & Wilson, J. (2012). Seven Databases in Seven Weeks.

[37] DB-Engines. DB-Engines Ranking - popularity ranking of key-value stores. http://db-engines.com/en/ranking/key-value+store. Accesado el 1 de Abril del 2014

[38] MongoDB. (2013). MongoDB. http://www.mongodb.org/. Accesado el 20 de Febrero del 2014.

[39] Cassandra. (2009). The apache Cassandra project. http://cassandra.apache.org/. Accesado el 20 de Febrero del 2014.

[40] OrientDB. (2014). OrientDB | Orient Technologies. http://www.orientechnologies.com/orientdb/. Accesado el 20 de Febrero del 2014.

[41] Neo4j. (2014). Neo4j - The World's Leading Graph Database. http://www.neo4j.org/. Accesado el 20 de Febrero del 2014.

[42] Redis. (2014). Redis. http://redis.io/. Accesado el 1 de Abril del 2014.

[43] Halpin, T. (2006). Object-role modeling (ORM/NIAM). In Handbook on architectures of information systems (pp. 81-103). Springer Berlin Heidelberg.

[44] Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009, March). A classification of object-relational impedance mismatch. In Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on (pp. 36-43). IEEE.

[45] ZODB. (2011). ZODB - a native object database for Python. http://www.zodb.org/en/latest/. Accesado el 20 de Febrero del 2014.

[46] Vagrant. (2013). Vagrant Documentation. http://docs.vagrantup.com/v2/. Accesado el 25 de Marzo del 2014.

[47] GeoNames. (2010). GeoNames. http://www.geonames.org/. Accesado el 25 de Marzo del 2014.

[48] GeoNames. (2014). GeoNames MX.zip. http://download.geonames.org/export/zip/MX.zip. Accesado el 25 de Marzo del 2014.

[49] GeoNames. (2014). Readme for GeoNames.org Postal Code files. http://download.geonames.org/export/zip/. Accesado el 25 de Marzo del 2014.

[50] MongoDB. (2014). Model Tree Structures with an Array of Ancestors. http://docs.mongodb.org/manual/tutorial/model-tree-structures-with-ancestors-array/. Accesado el 26 de Marzo del 2014.

[51] MongoDB. (2014). Read Operations. http://docs.mongodb.org/manual/core/read-operations/. Accesado el 26 de Marzo del 2014.

[52] MongoDB. (2014). Read Operations Overview. http://docs.mongodb.org/manual/core/read-operations-introduction/#projections. Accesado el 26 de Marzo del 2014.

[53] MongoDB. (2014). Cursors. http://docs.mongodb.org/manual/core/cursors/. Accesado el 26 de Marzo del 2014.

[54] Singh, A. (2013). Instant Cassandra Query Language. Packt Publishing Ltd.

[55] ebay Tech Blog. (Julio, 2012). Cassandra Data Modeling Best Practices, Part 1. http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/#.U1bAGlSx22g. Accesado el 2 de Abril del 2014.

[56] Datastax. (2014). Using collections. http://www.datastax.com/documentation/cql/3.0/cql/cql_using/use_collections_c.html. Accesado el 2 d Abril del 2014.

[57] Datastax. (2014). cqlsh. http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/cqlsh.html Accesado el 2 de Abril del 2014.

[58] Datastax. (2014). Using the list type - DataStax CQL 3.0 Documentation. http://www.datastax.com/documentation/cql/3.0/cql/cql_using/use_list_t.html. Accesado el 2 de Abril de 2014.

[59] Datastax Corporation. (Febrero, 2013). Benchmarking Top NoSQL Databases. http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf

[60] The apache software foundation. (Febrero, 2014). Cassandra 4511 - Secondary index support for CQL3 collections- ASF Jira. https://issues.apache.org/jira/browse/CASSANDRA-4511. Accesado el 3 de Abril del 2014.

[61] Orientechnologies (Mayo, 2014). SQL - Query https://github.com/orientechnologies/orientdb/wiki/SQL-Query. Accesado el 20 de Mayo del 2014

[62] Orientechnologies (Abril, 2014). SQL. https://github.com/orientechnologies/orientdb/wiki/SQL. Accesado el 20 de Mayo del 2014.

[63] Neo4j - The World's Leading Graph Database (2014) Learn Cypher. http://www.neo4j.org/learn/cypher. Accesado el 5 de Mayo del 2014.

[64] ar, S., & Agrawal, R. (1993). Extending SQL with generalized transitive closure. Knowledge and Data Engineering, IEEE Transactions on, 5(5), 799-812.

[65] Redis. Data Types. http://redis.io/topics/data-types. Accesado el 5 de Mayo del 2014.

[66] Atkinson, M. P., Bancilhon, F., DeWitt, D. J., Dittrich, K. R., Maier, D., & Zdonik, S. B. (Diciembr 1989). The Object-Oriented Database System Manifesto. In DOOD (Vol. 89, pp. 40-57).

[67] Caesar0301 - GitHub.(Mayo 2014). pyTree. https://github.com/caesar0301/pyTree. Accesado el 20 de Mayo del 2014

[68] AgusRumayor - GitHub (Mayo 2014). pyTree. https://github.com/AgusRumayor/pyTree. Accesado el 20 de Mayo del 2014

[69] About.com Linux. (2014). time. http://linux.about.com/library/cmd/blcmdl1_time.htm. Accesado el 20 de Mayo del 2014.

[70] Intel Developer Zone. (Febrero, 2014). Detection Disk I/O-bound Applications in Server Systems. https://software.intel.com/en-us/blogs/2014/02/21/detecting-disk-io-bound-applications-in-server-systems. Accesado el 20 de Mayo del 2014.

[71] Brewer, E. A. (2000, July). Towards robust distributed systems. En PODC (p. 7).

[72] Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51-59.

[73] Smith, J., & Nair, R. (2005). Virtual machines: versatile platforms for systems and processes Elsevier.

[74] Oracle VM VirtualBox. (2014). VirtualBox. https://www.virtualbox.org/. Accesado el 20 de Junio de 2014