UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

“Implementación de restricciones de integridad en una base de datos NoSQL documental”

TESIS

Que para optar por el grado de:

Maestro en Ingeniería de la computación

Presenta:
Ing. Diego Alonso Valdez Benítez

Director de tesis:
Dr. Jorge Luis Ortega Arjona
Facultad de Ciencias

México, D. F.

IIMAS
Noviembre 2015

Resumen

El modelo relacional de las bases de datos se ocupa con mucha frecuencia, pues está fundado sobre bases matemáticas sólidas, y se puede representar usando algoritmos computacionales. Esto posibilita una alta oferta de los sistemas manejadores de bases de datos relacionales, hecho que a su vez ha llevado a los desarrolladores a utilizarlo en sus aplicaciones.
Actualmente Oracle y otros RDBMS (*Relational Database Management System,* sistema manejador de base de datos relacional) son utilizados en muchas aplicaciones comerciales y por lo regular es común encontrarlos en empresas de cualquier sector. Dichos sistemas implementan el lenguaje estructurado de consultas (*Structured Query Language* o SQL).
Sin embargo, las tecnologías basadas en SQL son insuficientes para resolver todos los problemas de almacenamiento de datos. En algunos casos se han vuelto evidentes sus limitaciones: el escalamiento de datos, costosas lecturas de datos, el rendimiento de servidores singulares, y un esquema de almacenamiento rígido.
Como respuesta a dichas limitantes, ha surgido otro paradigma: el Not Only SQL (NoSQL). Este grupo de tecnologías ofrece cómputo de alto rendimiento y manejo de una gran cantidad de datos, además de escalabilidad y tolerancia a la partición. Puede manejar adecuadamente exabytes de datos, y es el preferido de empresas que manejan grandes volúmenes de información, como Google y Amazon . Lo anterior no significa que las bases de datos NoSQL son un reemplazo de las relacionales. En realidad, son una alternativa de almacenamiento de datos.
Sin embargo, los DBMS basados en NoSQL enfrentan otros problemas, como la carencia de control de integridad de los datos. Esto es, si se desean implementar restricciones en una base de datos no relacional, éstas deben ser programadas manualmente, pues no es una función de éstas.
Como respuesta a esta limitación, la presente tesis propone el desarrollo de estrategias de control de integridad de información en una base de datos documental, que es un subconjunto de las tecnologías NoSQL. Adicionalmente se procura que dicha implementación sea lo más eficiente posible, en términos de tiempo de respuesta.

Índice general

Índice de figuras 6
Índice de tablas 8

1. Introducción 9
   1.1. El Contexto……9
   1.2. El Problema……9
   1.3. La Hipótesis……9
   1.4. El Objetivo……10
   1.5. La Metodología……10
   1.6. Las Contribuciones……10
   1.7. Estructura de la tesis……...10

2. Antecedentes 12
   2.1. Introducción a las bases de datos……12
      2.1.1. Conceptos fundamentales de bases de datos. ……..12
      2.1.2. Objetivos de un DBMS……..……13
      2.1.3. Comandos y componentes de un DBMS……14
      2.1.4. Modelo de datos……...17
   2.2. Bases de datos relacionales. 19
      2.2.1. Estructura de datos en el modelo relacional. ……...19
      2.2.2. Restricciones de datos en el modelo relacional……20
      2.2.3. Operaciones en el modelo relacional…….....22
   2.3. Bases de datos no relacionales. 24
      2.3.1. Bases de datos distribuidas……...25
      2.3.2. Limitaciones de las bases de datos relacionales……...26
      2.3.3. Ventajas y desventajas de NoSQL……....27
      2.3.4. Teorema de CAP y propiedades BASE……28
      2.3.5. Tipos de DBMS no relacionales……...29
      2.3.6. Escalamiento de una base de datos no relacional……35
      2.3.7. Procesamiento de datos distribuido mediante MapReduce……..35
   2.4. Indexación 37
      2.4.1. Conceptos básicos…….....37
      2.4.2. Índices ordenados……38
      2.4.3. Índices de archivos mediante árboles B+-……....39
      2.4.4. Índices en MongoDB……41
   2.5. Resumen. 43

3. Trabajos Relacionados 44
   3.1. Integridad referencial en bases de datos NoSQL en la nube. 44
      3.1.1. Solución 1. Metadatos en súper columnas……...46
      3.1.2. Solución 2. Metadatos como un registro superior……46
      3.1.3. Solución 3. Familia de columnas de metadatos……48
      3.1.4. Solución 4. Clúster de metadatos……..49
      3.1.5. Implementación de las restricciones de integridad en un DBMS NoSQL…49
   3.2. Soporte de SQL sobre MongoDB usando metadatos. 50

3.2.1. Generalidades de MongoDB……50
3.2.2. Método propuesto……51
3.3. Integridad referencial y dependencias entre documentos en una base de datos orientada a documentos 53
3.3.1. MapReduce sobre MongoDB……..54
3.3.2. Verificación de relaciones uno a muchos……55
3.3.3. Verificación de relaciones muchos a muchos……..57
3.3.4. Conclusiones……58
3.4. Resumen. 59

4. Implementación de restricciones de integridad en una base de datos NoSQL documental 60
4.1. Descripción general de las estrategias. 60
4.2. Representación de metadatos. 61
4.2.1. Metadatos de un conjunto de restricciones……...61
4.2.2. Metadatos de restricción por conjunto de valores……62
4.2.3. Metadatos para la restricción de campos obligatorios……..63
4.3. Indexación de metadatos. 64
4.3.1. Justificación e implicaciones de la indexación……64
4.3.2. Descripción de los índices de metadatos……64
4.4. Implementación de restricciones de integridad de dominio 65
4.4.1. La interfaz de integridad……..65
4.4.2. Algoritmo principal de la interfaz de integridad……..66
4.4.3. Funciones de la interfaz de integridad……67
4.4.4. Inserción de metadatos……68
4.4.5. Inserción de documentos……69
4.4.6. Actualización de documentos……..71
4.4.7. Inserción y actualización sin verificación……....72
4.4.8. Eliminación de documentos……72
4.5. Optimizaciones de la interfaz de integridad 72
4.5.1. Necesidad de la optimización……...72
4.5.2. Paralelismo de la interfaz de integridad……...73
4.5.3. El caché de metadatos……..74
4.5.4. El algoritmo principal optimizado……..……75
4.5.5. Metadatos locales……77
4.6. Resumen 77

5. Casos de estudio 78
5.1. La base de datos de TPC-E. 78
5.2. La base de datos de TPC-H. 80
5.3. Diseño e implementación del experimento. 81
5.3.1. Metodología de pruebas de desempeño de la interfaz de integridad……82
5.3.2. Comprobación del funcionamiento correcto de la interfaz de integridad…83
5.4. Resultados de desempeño. 84
5.4.1. Resultados de los experimentos con datos de TCP-E……..85
5.4.2. Resultados de los experimentos con datos de TPC-H……99
5.5. Resumen. 110

6. Análisis y conclusiones 111
6.1. Resumen del trabajo……...111
6.2. Re-enunciado de la hipótesis…….111
6.3. Análisis de los resultados de TPC-E…….111
6.4. Análisis de los resultados de TPC-H……112
6.5. Conclusiones……112
6.6. Contribuciones…….113
6.7. Trabajos futuros……...113

7. Bibliografía y referencias 114

8. Apéndices 116

Apéndice A: Tablas y restricciones de TPC-E……116
Apéndice B: Tablas y restricciones de TPC-H……117

Índice de figuras

2.1. Entorno de un sistema de base de datos [8]……13
2.2. Interacción de los componentes de un DBMS [9]……..15
2.3. Clasificación de los modelos de datos según el enfoque [12]……18
2.4. Clasificación de los modelos de datos según el modo de almacenamiento. [12]…18
2.5. Base de datos centralizada en una red de computadoras [21]…….26
2.6. Entorno de un DDBS [21]……..……..……..……..……..……26
2.7. Comparación entre el almacenamiento de filas y el almacenamiento de columnas [20] …….30
2.8. Una Columna en Cassandra [25]……..31
2.9 Una súper columna en Cassandra [25] ……..31
2.10. Una familia de columnas en Cassandra [25]……...31
2.11. Un documento en MongoDB [2]…….32
2.12. Una colección en MongoDB. [2] …….33
2.13. Ejecución de MapReduce [13] ……...36
2.14. Aplicación de MapReduce a una base de datos [13]……...37
2.15. Archivo secuencial para registros de cuentas [27]…….39
2.16. Estructura de un nodo en un árbol B+- [27]……..……40
2.17. Un nodo hoja para el índice de cuenta en un árbol B+- [27].……40
2.18. Árbol B+-para el archivo de cuentas [27]……...41
2.19. Diagrama de una consulta que utiliza un índice [28]……..41
2.20. Diagrama de un índice compuesto [29]…….....42
3.1. Ejemplo del almacenamiento de metadatos [24]……...45
3.2. Almacenamiento de metadatos en súper columnas [24]……...46
3.3. Almacenamiento de metadatos en un registro superior [24]…….47
3.4. Almacenamiento de metadatos en familias de columnas [24]……..48
3.5. Almacenamiento de metadatos en un clúster separado del keyspace [24]……49
3.6. Arquitectura de MongoDB [25]…….51
3.7. Arquitectura del sistema propuesto [25]……52
3.8. Diagrama de flujo del sistema [25]……53
4.1. Estructura de un documento de metadatos para un conjunto de restricciones…...61
4.2. Estructura de un documento de metadatos para una restricción por conjunto de valores……62
4.3. Estructura de un documento de metadatos para una restricción de campos obligatorios……63
4.4. Diagrama de flujo de datos entre la aplicación, la interfaz y la base de datos…...65
4.5. Diagrama de flujo del algoritmo principal de la interfaz de integridad……66
4.6. Ejemplo del contenido del caché implementado como tabla hash……74
5.1. Flujo de transacciones en el modelo de negocio [22]……78
5.2. Componentes de la aplicación [22]……79
5.3. El entorno de negocio de TPC-H [23]……80
5.4. Esquema de la base de datos TPC-H [23]……...81
5.5. Comprobación del comportamiento correcto de la interfaz de integridad mediante consultas……84
5.6. Gráfica de tiempo promedio del experimento 1 (TPC-E).……..86
5.7. Comprobación del funcionamiento de la interfaz de integridad……86
5.8. Gráfica de tiempo promedio del experimento 2 (TPC-E).……..87

5.9. Gráfica de tiempo promedio del experimento 3 (TPC-E)……..88
5.10. Comprobación del funcionamiento de la interfaz de integridad……89
5.11. Gráfica de tiempo promedio del experimento 4 (TPC-E)……90
5.12. Gráfica de tiempo promedio del experimento 5 (TPC-E)……91
5.13. Gráfica de tiempo promedio del experimento 6 (TPC-E)……92
5.14. Gráfica de tiempo promedio del experimento 7 (TPC-E)……93
5.15. Comprobación del funcionamiento de la interfaz de integridad……..94
5.16. Gráfica de tiempo promedio del experimento 8 (TPC-E)……95
5.17. Gráfica de tiempo promedio del experimento 9 (TPC-E)……96
5.18. Gráfica de tiempo promedio del experimento 10 (TPC-E)…….….97
5.19. Gráfica de tiempo promedio del experimento 11 (TPC-E)…….…98
5.20. Gráfica de tiempo promedio del experimento 12 (TPC-E)…….…99
5.21. Gráfica de tiempo promedio del experimento 1 (TPC-H)……...100
5.22. Comprobación del funcionamiento de la interfaz de integridad……..100
5.23. Gráfica de tiempo promedio del experimento 2 (TPC-H)……...101
5.24. Gráfica de tiempo promedio del experimento 3 (TPC-H)……...102
5.25. Comprobación del funcionamiento de la interfaz de integridad……..103
5.26. Gráfica de tiempo promedio del experimento 4 (TPC-H)……...104
5.27. Gráfica de tiempo promedio del experimento 5 (TPC-H)……...104
5.28. Comprobación del funcionamiento de la interfaz de integridad……..105
5.29. Gráfica de tiempo promedio del experimento 6 (TPC-H)……...106
5.30. Gráfica de tiempo promedio del experimento 7 (TPC-H)……...107
5.31. Gráfica de tiempo promedio del experimento 8 (TPC-H)……...108
5.32. Gráfica de tiempo promedio del experimento 9 (TPC-H)……...109
5.33. Gráfica de tiempo promedio del experimento 10 (TPC-H)…….109

Índice de tablas

2.1. Ejemplo de una tabla en el modelo relacional……20
2.2. Tabla de alumnos con las materias que cursan……...22
2.3. Comparación entre los distintos tipos de DBMS [13].……...33
2.4. Comparación entre distintos DBMS NoSQL [36]……34
5.1. Resultados de desempeño del experimento 1 (TPC-E) ……85
5.2. Resultados de desempeño del experimento 2 (TPC-E) ……87
5.3. Resultados de desempeño del experimento 3 (TPC-E) ……88
5.4. Resultados de desempeño del experimento 4 (TPC-E) ……89
5.5. Resultados de desempeño del experimento 5 (TPC-E) ……91
5.6. Resultados de desempeño del experimento 6 (TPC-E) ……92
5.7. Resultados de desempeño del experimento 7 (TPC-E) ……93
5.8. Resultados de desempeño del experimento 8 (TPC-E) ……94
5.9. Resultados de desempeño del experimento 9 (TPC-E) ……95
5.10. Resultados de desempeño del experimento 10 (TPC-E) ……96
5.11. Resultados de desempeño del experimento 11 (TPC-E) ……97
5.12. Resultados de desempeño del experimento 12 (TPC-E) ……98
5.13. Resultados de desempeño del experimento 1 (TPC-H) ……99
5.14. Resultados de desempeño del experimento 2 (TPC-H) ……101
5.15. Resultados de desempeño del experimento 3 (TPC-H) ……102
5.16. Resultados de desempeño del experimento 4 (TPC-H) ……103
5.17. Resultados de desempeño del experimento 5 (TPC-H) ……104
5.18. Resultados de desempeño del experimento 6 (TPC-H) ……105
5.19. Resultados de desempeño del experimento 7 (TPC-H) ……106
5.20. Resultados de desempeño del experimento 8 (TPC-H) ……107
5.21. Resultados de desempeño del experimento 9 (TPC-H) ……108
5.22. Resultados de desempeño del experimento 10 (TPC-H) ……109
A-1. Restricciones para la tabla CUSTOMER_ACCOUNT de la base de datos TPC-E……116
A-2. Restricciones para la tabla LAST_TRADE de la base de datos TPC-E……...116
A-3. Restricciones para la tabla DAILY_MARKET de la base de datos TPC-E…….116
A-4. Restricciones para la tabla COMPANY de la base de datos TPC-E……116
B-1. Restricciones para la tabla PART de la base de datos TPC-H……..117
B-2. Restricciones para la tabla SUPPLIER de la base de datos TPC-H……..117
B-3. Restricciones para la tabla CUSTOMER de la base de datos TPC-H……..117

Capítulo 1. Introducción

1.1. El Contexto

Las bases de datos NoSQL surgen como respuesta a la demanda creciente de almacenamiento de grandes volúmenes de información (normalmente del orden de terabytes en adelante), frecuencia de acceso a los datos y necesidades superiores de rendimiento y procesamiento. En contraste, las bases de datos relacionales son insuficientes para superar los retos de escalabilidad y agilidad que exigen algunas aplicaciones modernas [3] [4]. Por este motivo muchas aplicaciones utilizan enfoques alternativos en busca del alto rendimiento, alta disponibilidad y escalamiento masivo que ofrece NoSQL [4]. A cambio de estos beneficios, las bases de datos NoSQL carecen por completo de control de integridad y el manejo de transacciones es menos estricto [5].

Las bases de datos relacionales y las bases de datos NoSQL no son tecnologías completamente opuestas, y de hecho, en algunos casos se han combinado en sistemas híbridos que tienen características de los dos enfoques. Ejemplos de esto incluyen CloudTPS de Google BitTable y Amazon SimpleDB [5]. Los tipos de DBMS NoSQL incluyen las bases de datos documentales, el almacenamiento en grafos, almacenamiento clave-valor y almacenamiento de columnas [3].

Esta tesis se enfoca exclusivamente en las bases de datos documentales, pues de acuerdo con estudios su rendimiento, escalabilidad y flexibilidad de almacenamiento tienden a ser altos. Además tienen baja complejidad de implementación [13].

1.2. El Problema

En los DBMS NoSQL hacen falta mecanismos de especificación y control de restricciones que aseguren la integridad de datos insertados y actualizados [6]. La implementación de esto requiere de programación manual y externa a la base de datos. El problema en esta tesis se delimita a las bases de datos documentales y a restricciones de dominio en un ambiente cliente-servidor.

1.3. La Hipótesis

Dada una base de datos documental, ¿Es posible implementar externamente el procesamiento de restricciones de integridad de dominio de tal forma que la validación de los documentos sea una tarea cuyo tiempo de respuesta crezca directamente proporcional a la cantidad de documentos a verificar?
Por *integridad* se entiende validez y exactitud de la información en términos de restricciones definidas en la base de datos. Un documento verificado se considera íntegro.

1.4. El objetivo

Se busca la obtención de un software que permita implementar restricciones de integridad de dominio en una base de datos NoSQL documental. Dicho software debe ser tan eficiente como sea posible, en términos de tiempo de respuesta.

1.5. La Metodología

Para dar solución al problema, se plantea la especificación de metadatos como documentos dentro de una colección, que son usados para validar la integridad de documentos de datos que se insertan en dicha colección. Una capa de software intermedia entre la base de datos y la aplicación se encarga de hacer estas validaciones. Este software aplica las inserciones y actualizaciones que se consideren correctas desde el punto de vista de las restricciones previamente especificadas en los documentos de metadatos.

El requerimiento no funcional que se refiere a la proporción directa del tiempo de respuesta respecto del volumen de datos a procesar, se implementa utilizando características de optimización que incluyen indexación de metadatos, programación concurrente, y desarrollo de un caché lógico de metadatos en la máquina cliente.

1.6. Las Contribuciones

1. Se proveen de estrategias que permiten la especificación y control de restricciones de integridad de manera eficiente en bases de datos orientadas a documentos. Esto se traduce en los algoritmos necesarios para implementar dicho control y de un modelo de metadatos.
2. Se aportan también características de optimización que permiten minimizar el tiempo de respuesta.

1.7. Estructura de la tesis

- En el capítulo 2 se presenta el marco teórico sobre las bases de datos. Esto incluye los conceptos básicos de bases de datos, características del modelo relacional, propiedades de las bases de datos NoSQL y teoría de indexación con ejemplos en el sistema de MongoDB.

- En el capítulo 3 se revisan las investigaciones previas relativas al control de integridad en bases de datos NoSQL. Este apartado comprende tres investigaciones donde se trabaja con tecnología de bases de datos no relacionales con fines muy similares a los de esta tesis. Dos de estos trabajos se enfocan en la implementación de integridad referencial que extiende la funcionalidad de bases de datos NoSQL, mientras que en esta tesis el enfoque es hacia la integridad de dominio.

- En el capítulo 4 se describen en detalle los algoritmos de control de integridad propuestos y las características de optimización. También se explican con precisión el modelo de metadatos utilizado y el papel que juega la indexación.

- En el capítulo 5 se valida la hipótesis mediante experimentos con bases de datos en un ambiente cliente-servidor y se mide el desempeño del software desarrollado. Aquí se presentan todos los resultados de desempeño obtenidos en las pruebas que se hicieron con datos estandarizados de los benchmarks TPC-E y TPC-H.

- En el capítulo 6 se analizan los resultados de tiempos de respuesta obtenidos en la fase de pruebas de desempeño. Se responde con precisión y en detalle a la pregunta de hipótesis.

Capítulo 2. Antecedentes

En este capítulo se introducen conceptos básicos de las bases de datos. También se abordan las características principales del modelo relacional y se describen en detalle las propiedades comunes a todas las tecnologías de NoSQL. Una sección adicional describe los principios de indexación y algunas de sus técnicas.

2.1 Introducción a las bases de datos

2.1.1 Conceptos fundamentales de bases de datos

Una *base de datos* es una colección de datos relacionados que existe a lo largo de un período de tiempo. *Datos* se refiere a hechos conocidos que se pueden almacenar y tienen algún significado implícito [8].

Las propiedades implícitas de las bases de datos son [8]:

- Una base de datos representa algún aspecto del mundo real.
- Una base de datos es una colección de datos lógicamente coherente con algún tipo de significado inherente.
- Una base de datos se diseña, construye y llena con datos para un propósito específico. Dispone de un grupo pretendido de usuarios y algunas aplicaciones que utilizan estos usuarios.

Una base de datos puede tener cualquier tamaño y complejidad. Puede ser gestionada manualmente o de forma computarizada [8].

Las bases de datos son administradas por un *sistema administrador de base de datos* (DBMS, database management system) o sistema manejador de base de datos.
El DMBS es una colección de programas que permite la creación y mantenimiento de una base de datos. Es un sistema que facilita los procesos de definición, construcción, manipulación y compartición de bases de datos entre varios usuarios y aplicaciones. Les da a los usuarios la habilidad de consultar y modificar datos [8].

Definir una base de datos implica especificar los tipos de datos, estructuras y restricciones de los datos que se almacenan en la base de datos. La *definición* o información descriptiva de una base de datos también se almacena en esta última, en forma de catálogo o diccionario, y es lo que comúnmente se conoce como *metadatos* [8].

Una aplicación accede a la base de datos enviando consultas de datos al DMBS. Una *consulta* o *query* provoca la recuperación de algunos datos; una *transacción* puede provocar la lectura o la escritura de algunos datos. Un *sistema de base de datos* es la combinación de una base de datos y el DBMS que se utiliza para administrarla [8].

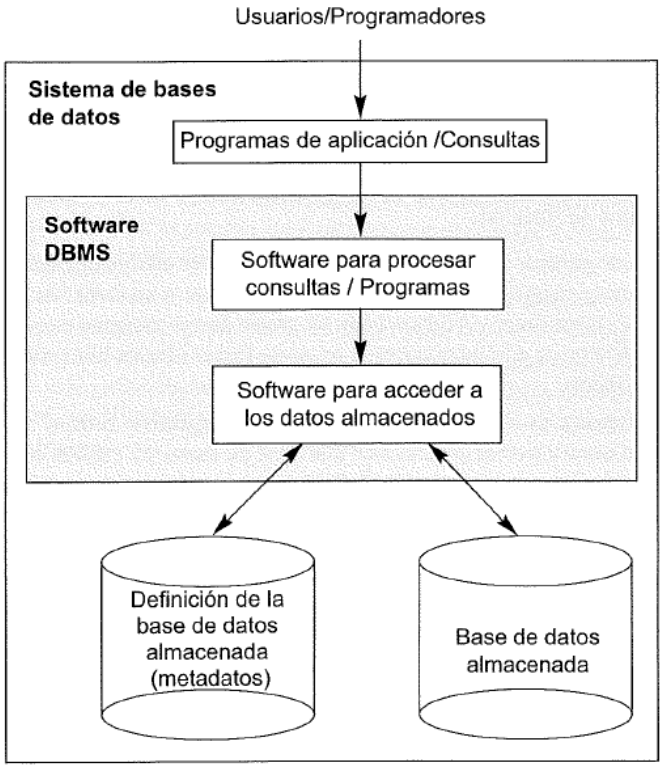

Figura 2.1. Entorno de un sistema de base de datos [8].

La figura 2.1 ilustra el entorno de un sistema de base de datos. Los usuarios y/o programadores utilizan un programa de aplicación que interactúa con el DBMS, y éste se encarga de procesar la consulta, acceder a los datos solicitados, y devolver a la aplicación los resultados esperados [8]. La definición de la base de datos se almacena en el catálogo del DBMS y contiene información como la estructura de cada archivo, el tipo y el formato de almacenamiento de cada elemento de datos, y distintas restricciones de los datos. Esta información se denomina *metadatos*. El DBMS utiliza el catálogo cuando necesita información sobre la estructura de la base de datos [8].

2.1.2. Objetivos de un DBMS

Todo DBMS debe perseguir los siguientes objetivos [11]:

- **Abstracción de la información.** Para las aplicaciones es transparente cómo el DBMS tiene almacenada físicamente la información.

- **Independencia**. Los programas de aplicación que utilizan los usuarios para hacer consultas a la base de datos no dependen de cómo está físicamente almacenada la información ni de cómo está organizada y relacionada internamente la información. Entonces los esquemas físico y lógico de la base de datos pueden alterarse sin impactar a las aplicaciones.

- **Seguridad**. Existe un sistema de permisos que les da a los usuarios y grupos de usuarios ciertos privilegios de acceso a la base de datos.

- **Integridad**. El DBMS debe garantizar la validez de los datos almacenados contra cualquier circunstancia que sea capaz de corromper la información.

- **Respaldo y recuperación**. Se debe proporcionar una forma eficiente de realizar copias de respaldo de la información almacenada de tal forma que se pueda restaurar el estado de la base de datos a partir de estas copias en caso de pérdida.

- **Control de concurrencia**. El acceso y alteración simultáneos de los datos debe controlarse para proteger la consistencia de los mismos.

2.1.3. Comandos y componentes de un DBMS

El sistema manejador de la base de datos distingue entre dos tipos principales de comandos [9]:

1. Aquéllos que vienen de usuarios y aplicaciones que consultan datos.
2. Aquéllos que usa un administrador de la base de datos, quien es responsable de estructurar y esquematizar la base de datos.

Los comandos que usan los administradores de una base de datos tienen que ver con darle estructura a la misma, establecer las relaciones entre tablas y definir las restricciones de los datos. Ésto también incluye comandos para la creación, edición, eliminación de tablas y determinar los permisos de los usuarios. En conjunto toda esta familia de comandos se llama *Data Definition Language* (DDL). Estos comandos son verificados por un procesador DDL, luego se envían al motor de ejecución, el cual a su vez accede al administrador de índices para alterar los metadatos. Esto es el esquema de la base de datos [9].
Otro tipo de comandos son los de usuario y no pueden afectar el esquema de la base de datos, sino alterar los contenidos de la misma. En general, éstas son inserciones, eliminaciones o alteraciones de registros. A este conjunto de comandos se le conoce como
Data Manipulation Language (DML). La mayoría de las interacciones con un DBMS son hechas por un usuario que usa los comandos DML [9].
Cuando un usuario realiza una consulta, ésta es procesada por un *compilador de consultas (query compiler),* el cual define la secuencia de acciones que el DBMS debe hacer. Estos datos se envían al *motor de ejecución (execution engine)*, el cual hace solicitudes de ciertas piezas de datos llamadas *registros* a un *administrador de recursos (resource manager).* Este componente guarda información sobre en qué archivos están localizados los registros, con qué formato y cuáles archivos de índice se necesitan para acceder a dichos registros. Los archivos de índice permiten encontrar rápidamente los datos en los archivos [9].
La solicitud luego llega al *administrador de buffer (buffer manager)* que permite traer porciones de datos del disco duro a la memoria principal, a través del *administrador de almacenamiento (storage buffer)*. Las unidades de almacenamiento pueden ser páginas o bloques de disco. El administrador de almacenamiento puede requerir de comandos del

sistema operativo para acceder al disco duro, o bien esto puede ser controlado por el DBMS [9].
En la figura 2.2 se presenta una vista completa de los componentes que interactúan dentro de un DBMS, y los datos que se transfieren entre ellos.

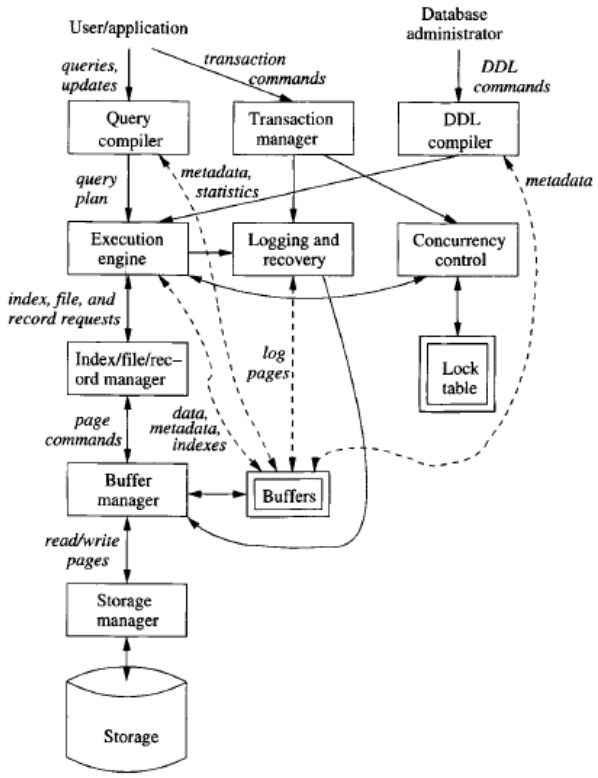

Figura 2.2. Interacción de los componentes de un DBMS [9].

Administración de almacenamiento y de buffer

Los datos de una base de datos normalmente residen en almacenamiento secundario, que por lo general es un disco duro. Sin embargo, para realizar cualquier operación útil, es necesario que los datos se encuentren en la memoria principal, y es trabajo del administrador de almacenamiento colocar los datos del disco en la memoria y viceversa.
En un sistema de base de datos sencillo, el administrador de almacenamiento no es más que el sistema de archivos del sistema operativo. Pero por motivos de eficiencia, el DBMS controla directamente el almacenamiento en disco, al menos bajo ciertas circunstancias.
El administrador de almacenamiento lleva un registro de la localización de los archivos en el disco y obtiene los bloques que contienen el archivo en respuesta a la solicitud del administrador de buffer [9].
El administrador de buffer se encarga de segmentar la memoria principal en buffers, que son regiones en las cuales se pueden transferir bloques de disco. Todos los componentes de un DBMS que necesiten datos del disco, primero deben interactuar con los buffers y el administrador de buffer, que puede ser directamente o mediante el motor de ejecución.

Los tipos de información que varios componentes pueden necesitar son [9]:

- **Datos**: contenidos de la base de datos.
- **Metadatos**: el esquema de la base de datos que define su estructura y restricciones.
- **Registros log**: información sobre los cambios recientes en la base de datos.
- **Estadísticas**: información reunida y almacenada por el DBMS sobre las propiedades de los datos.
- **Índices**: estructuras de datos que soportan acceso eficiente a los datos.

Procesamiento de transacciones

Es normal agrupar varias operaciones de una base de datos en una transacción, la cual es atómica, consistente, aislable y durable. Estas características se abrevian comúnmente como ACID (*Atomicity, Consistency, Isolation, Durability*). La *atomicidad* asegura que una transacción tiene éxito o falla, sin resultados parciales. La *consistencia* significa que la base de datos se encuentra en un estado conocido y deseable, antes y después de una transacción. El *aislamiento* significa que las transacciones son independientes una de otra, y por lo tanto mutuamente excluyentes si operan sobre los mismos datos. Finalmente la *durabilidad* implica que los resultados de una transacción perduran incluso si el sistema falla después de completada la transacción [27].

El administrador de transacciones acepta comandos de transacción de una aplicación, la cual le indica en donde comienzan y terminan las transacciones. El procesador de transacciones ejecuta las siguientes tareas [9]:

- **Logging:** Para asegurar durabilidad en la base de datos, todo cambio en la base de datos se guarda como un log separado en el disco. El *administrador de log (log manager)* sigue ciertas políticas que aseguran que sin importar si hay fallas en el sistema, el *administrador de recuperación* (*recovery manager*) puede examinar los logs y restaurar a la base de datos a un estado consistente.
  El administrador de log inicialmente escribe el log en buffers e interactúa con el administrador de buffer para asegurar que los buffers se escriban en el disco.

- **Control de concurrencia:** Las transacciones deben ejecutarse en aislamiento. Si hay muchas transacciones ejecutándose al mismo tiempo, el administrador de control de concurrencia (*scheduler*) asegura que las acciones individuales de cada transacción se ejecutan en un orden que tiene el mismo efecto que haber ejecutado una transacción a la vez. Normalmente esto se logra por *bloqueos* (*locks)* sobre ciertas piezas de datos en la base de datos.
  Estos bloqueos previenen que dos o más transacciones accedan a la misma pieza de datos al mismo tiempo. Los bloqueos se encuentran en una tabla de la memoria principal, y el scheduler los usa para prohibir que el motor de ejecución acceda a partes bloqueadas de la base de datos.

- **Resolución de deadlock:** El administrador de transacciones previene que dos transacciones que están compitiendo por recursos se bloqueen mutuamente e

indefinidamente por un recurso que necesitan de otra transacción. La operación de *rollback* o *abortar* puede cancelar transacciones permitiendo que otras procedan.

Procesamiento de consultas

Uno de los componentes de un DBMS que más afecta el rendimiento es el procesador de consultas. Éste se encuentra compuesto por [9]:

1. El *compilador de consultas*, que traduce la consulta en una forma interna llamada plan de ejecución, que es una secuencia de operaciones a realizar sobre los datos.
El compilador de consultas se compone de tres unidades:

   a) Un *parser* de consultas, que construye una estructura arborescente a partir de la forma textual del query.
   b) Un *preprocesador* de consultas, que ejecuta revisiones semánticas en la consulta y transforma el árbol construido por el parser en un árbol de operadores algebraicos que representa el plan inicial de consulta.
   c) Un *optimizador* de consultas, el cual transforma el plan inicial de consulta en la mejor secuencia de operaciones disponible sobre los datos.

   El compilador utiliza metadatos y estadísticas sobre los datos para decidir cuál secuencia de operaciones se ejecuta más rápido.

2. El *motor de ejecución*, que se encarga de ejecutar cada uno de los pasos del plan de ejecución. El motor de ejecución interactúa con la mayoría de los componentes del DBMS; ya sea directamente o mediante los buffers. Para manipular datos necesita acceder a la base de datos y tomarlos como buffers. Su interacción con el scheduler le impide acceder a datos bloqueados. Para asegurarse de que los cambios en la base de datos se guardaron debe interactuar con el log manager.

2.1.4. Modelo de datos

El concepto de modelo de datos es uno de los más fundamentales en el estudio de las bases de datos. Un modelo de datos es una notación para describir datos o información. Concretamente es el conjunto de conceptos, reglas y convenciones que permiten describir la representación de la información en términos de datos [9].

La descripción generalmente consiste de tres partes:

1. **Estructura de los datos.** Las estructuras de datos usadas para implementar datos en una computadora son referidas frecuentemente como el modelo físico de datos.
2. **Operaciones en los datos.** En las bases de datos existe un conjunto limitado de operaciones, clasificadas principalmente en consultas y modificaciones.
   Las consultas permiten obtener o extraer información de la base de datos mientras que las modificaciones son alteraciones a los datos de la misma.

3. **Restricciones en los datos**. Los modelos de datos usualmente tienen una manera de describir las limitaciones de los datos que pueden almacenar. Estas restricciones pueden ser muy simples o muy complejas.

Los modelos de datos se clasifican según el enfoque en externo, interno y global [12]. La figura 2.3 muestra los detalles de esta clasificación.

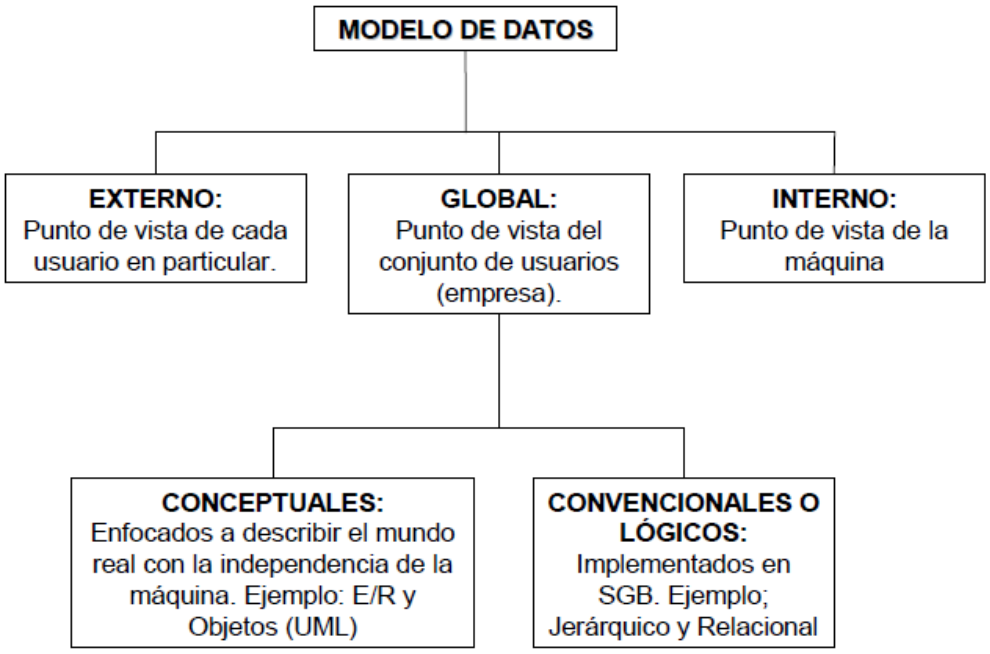

Figura. 2.3. Clasificación de los modelos de datos según el enfoque [12].

Una segunda clasificación de los modelos de datos es de acuerdo al modo de almacenamiento y organización de los datos. En el diagrama de la figura 2.4 se encuentran algunas de las categorías de esta clasificación.

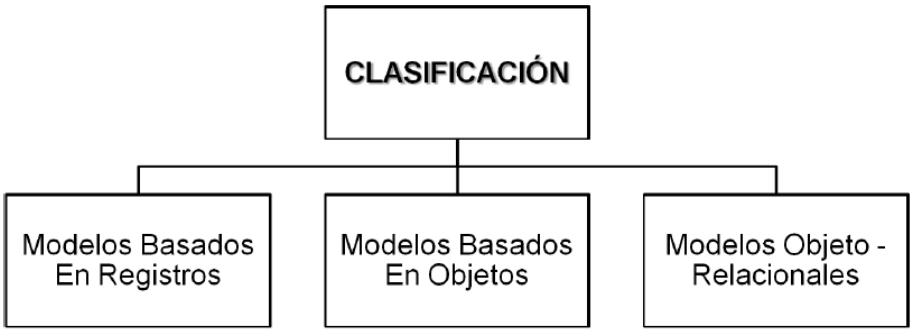

Figura 2.4. Clasificación de los modelos de datos según el modo de almacenamiento. [12]

Por ahora centramos nuestra atención en el modelo relacional, que es un modelo de datos convencional o lógico basado en registros. Es convencional porque su implementación es directa en un DBMS. Es basado en registros pues la información está almacenada como

estructuras abstractas representadas como un conjunto de campos de distinto tipo que refieren a un mismo individuo.

2.2. Bases de datos relacionales

El modelo relacional de las bases de datos se propuso por E.F. Codd en 1970, como una solución a los problemas que presentaban los modelos de datos hasta entonces existentes.
El modelo rompe con la dependencia entre datos y programa de aplicación, además de superar los problemas del esquema de almacenamiento de archivos como el manejo de la redundancia y de la consistencia [7].
El modelo relacional se funda sobre la teoría de conjuntos y la lógica de predicados para dar soporte a todas las operaciones en una base de datos. Codd también introduce la idea de un lenguaje de alto nivel que permitiera la recuperación de información de una base de datos [7].

El lenguaje SEQUEL (*Structured Query Language*) –mejor conocido como SQL -- se propone formalmente en 1974 por Donald D. Chamberlin y Raymond F. Boyce, como una implementación del modelo relacional y una mejora sobre el lenguaje de consultas SQUARE [10].

Las contribuciones de Codd y de Chamberlin y Boyce se vieron materializadas años más tarde en los primeros RDBMS.

2.2.1. Estructura de datos en el modelo relacional

En la implementación del modelo relacional, todos los datos se representan en tablas bidimensionales llamadas *relaciones*. Cada tabla está compuesta por filas llamadas *tuplas* y columnas llamadas *atributos*. Usualmente los atributos describen el significado de los datos que guarda esa columna [9].
El nombre de una relación y el conjunto de atributos asociado a dicha relación es lo que se conoce como el *esquema* de esa relación. En el modelo relacional, una base de datos está compuesta por una o más relaciones. El conjunto de esquemas para esas relaciones es el *esquema de la base de datos*. [9]
Las tuplas de una relación son todas las filas a excepción de la fila de encabezado que contiene los nombres de los atributos. Una tupla tiene un componente para cada atributo de una relación [9].

A continuación se introduce un ejemplo de una tabla de alumnos de licenciatura, donde:

- La entidad es Alumno.
- Los atributos son el número de cuenta, nombre, semestre y carrera.
- Existen tres tuplas, cada una con información de un alumno diferente.

ALUMNO

Número de cuenta	Nombre del alumno	Semestre	Carrera
28000	Juan Pérez	2	Ingeniería mecánica
41440	Ramiro Juárez	5	Diseño industrial
44835	Esteban López	7	Mercadotecnia

Tabla 2.1. Ejemplo de una tabla en el modelo relacional

El modelo relacional requiere que cada componente de una tupla sea *atómico*; esto es, debe ser de un tipo elemental como una cadena o un entero. No se permite que un valor guarde un conjunto, lista, arreglo u otro tipo de dato que pueda ser dividido en pequeños componentes [9].

Dominio. A cada atributo de una relación se le asocia un *dominio*, que es un tipo de dato elemental. Los componentes de una tupla en una relación deben tener, en cada componente, un valor que pertenece al dominio de la columna correspondiente. Por ejemplo, en la tabla de alumnos el número de cuenta y el semestre deben ser valores enteros positivos, mientras que el nombre del alumno y la carrea deben ser cadenas [9].

Orden de tuplas y atributos. Las relaciones son conjuntos de tuplas, no listas de tuplas. Entonces, el orden en que se encuentran las tuplas de una relación es irrelevante y no aporta información. Por ejemplo, se pueden invertir por completo las tuplas de la tabla de los alumnos y la información sigue siendo exactamente la misma [9].
El reordenamiento de los atributos también se puede hacer sin cambiar la relación, pero se debe hacer con cuidado. Si se cambia el orden de los atributos también cambia el orden de los componentes de las tuplas. De cualquier modo, un intercambio de las columnas tampoco afecta la relación. Por ejemplo, en la tabla de alumnos se pueden intercambiar las columnas de número de cuenta y nombre sin alterar la información de la relación [9].

Cambios en la relación y en el esquema. Una relación es dinámica o cambiante en el tiempo. Por ejemplo, en la tabla 2.1 es posible que se necesiten insertar nuevas tuplas de alumnos, borrar a los alumnos egresados o actualizar el valor del semestre. Lo que es menos probable que cambie es el esquema de la relación. Sin embargo, puede haber casos en los que sea necesario agregar o quitar atributos. Los cambios de esquema son muy costosos en grandes bases de datos, ya que requieren la reinserción de tuplas para añadir o borrar componentes [9].

2.2.2. Restricciones de datos en el modelo relacional

El modelo relacional permite definir ciertas restricciones respecto de los valores que pueden tomar los datos. Las restricciones que existen en una base de datos relacional se dividen en tres categorías principales [8]:

1. **Restricciones implícitas**: son aquéllas que son inherentes al modelo de datos.
2. **Restricciones explícitas:** son aquéllas que se pueden expresar directamente en los esquemas del modelo de datos, por lo general especificándolas en el DDL.

3. **Restricciones semánticas o basadas en aplicación**: son aquéllas que no se pueden expresar directamente en los esquemas del modelo de datos, y que por consiguiente deben ser expresadas e implementadas por los programas.

Las restricciones explícitas pueden ser de dominio, de clave, de valores NULL, de integridad de entidad y de integridad referencial.

Restricciones de dominio.

Las restricciones de dominio especifican que dentro de una tupla, el valor de un atributo *A* debe ser un valor atómico del dominio *dom(A).* Los tipos de datos asociados a ellos pueden ser numéricos, cadenas, valores lógicos, fechas, horas, etc. También es posible describir un dominio como un subconjunto de algún tipo de dato [8].

Restricciones de clave y de valor nulo.

Una relación exige que dos tuplas no puedan tener la misma combinación de valores para todos sus atributos. Un subconjunto de estos atributos en una relación en la cual no hay dos tuplas con la misma combinación de valores en sus atributos conforma una *superclave*, la cual especifica cierta restricción de exclusividad. Cada relación tiene al menos una superclave: el conjunto de todos sus atributos [8].
Una *clave* de un esquema de relación es una superclave que tiene la propiedad de que al eliminar cualquier atributo del conjunto clave, esto resulta en un conjunto de atributos que ya no es superclave de la relación [8]. Entonces una clave satisface que dos tuplas diferentes de la relación no pueden tener valores idénticos para todos los atributos de la clave. Además es una superclave mínima, es decir, una superclave de la cual no se puede eliminar ningún atributo y sigue teniendo la restricción de exclusividad [8].
El valor de un atributo clave puede usarse para identificar de forma única cada tupla en una relación. En general, un esquema de relación puede contar con más de una clave, y cada una de estas claves recibe el nombre de *clave candidata*. Arbitrariamente se puede elegir cualquiera de estas claves para ser designada como la *clave principal* o *clave primaria* de la relación [8]. Otra restricción en los atributos especifica si se permiten o no valores NULL.

Restricciones de integridad de entidad, integridad referencial y claves foráneas

Las restricciones de integridad de entidad declaran que el valor de ninguna clave principal puede ser nulo. Esto es debido a que dicha clave se usa para identificar tuplas individuales en una relación. Si este valor fuera nulo entonces no habría manera de identificar estas tuplas [8].
Las restricciones de clave y de integridad de entidad aplican a relaciones individuales.
Las de integridad referencial están especificadas entre dos relaciones y se utilizan para mantener la consistencia entre las tuplas de dos relaciones. Estas restricciones aseguran que una tupla de una relación que hace referencia a otra relación debe referir a una *tupla existente* en esa relación. Esto conduce al concepto de *clave foránea*. El propósito de una clave foránea es referenciar atributos de una relación con atributos de una segunda relación. La restricción de clave foránea asegura que un valor que aparece en una relación también debe aparecer como componente de la clave primaria de otra relación [8].

2.2.3. Operaciones en el modelo relacional

Las operaciones en el modelo relacional se clasifican en *recuperaciones* y *actualizaciones*. El álgebra relacional permite especificar recuperaciones. Una expresión de álgebra relacional conforma una nueva relación una vez aplicados una serie de operadores algebraicos a un conjunto de relaciones ya existente. Se basa en formular una consulta que especifica los datos que interesan mediante operadores relacionales. El resultado es una nueva relación que se convierte en la respuesta de la consulta formulada [8].

Existen tres tipos de modificaciones o escrituras en una base de datos relacional: *inserción*, *eliminación* y *actualización*. La inserción agrega nuevas tuplas a la relación, la eliminación borra tuplas existentes y la actualización cambia los valores de los atributos de ciertas tuplas. Siempre que se aplique cualquiera de estas operaciones, deben respetarse las restricciones de integridad especificadas en el esquema de bases de datos [8].

La operación de inserción. La inserción proporciona una lista de los valores de atributo para una nueva tupla que se inserta en una relación. Esta operación puede violar cualquiera de las cuatro restricciones antes mencionadas: las de dominio, si el valor dado a un atributo no aparece en el dominio correspondiente; las de clave, si el valor de dicha clave en la nueva tupla ya existe en otra tupla de la relación; las de integridad de entidad, si la clave principal de la nueva tupla es un nulo; y las de integridad referencial, si el valor de cualquier clave foránea en la nueva tupla se refiere a una tupla que no exista en la relación referenciada [8]. Considere la tabla 2.2, la cual relaciona los alumnos de la tabla 2.1 con las materias que cursan.

ALUMNO-MATERIA

Número de cuenta	**Clave de materia**	**Calificación**
28000	300	7
41440	400	8
44835	500	9

Tabla. 2.2. Tabla de alumnos con las materias que cursan

El número de cuenta y la clave de materia son atributos que juntos forman una clave primaria compuesta, de tal forma que dos tuplas no pueden tener los mismos valores en este par de atributos, de acuerdo con las restricciones de clave. El número de cuenta en esta tabla también es una clave foránea que apunta a la tupla correspondiente de la tabla ALUMNO.

Ahora suponga que se insertan las siguientes tuplas en la tabla 2.2:

- `INSERTAR EN ALUMNO-MATERIA (28000, 300, 9)`
  Esta operación viola la restricción de clave porque ya existe una tupla con los mismos valores de número de cuenta y clave de materia.

- `INSERTAR EN ALUMNO-MATERIA (NULL, 400, 8)`
  Esta operación viola la restricción de integridad de entidad porque un atributo de clave primaria no puede ser nulo.

- `INSERTAR EN ALUMNO-MATERIA (70800, 500, 10)`
  Esta operación viola la restricción de integridad referencial porque en la tabla 2.1 no existe ninguna tupla con el valor 70800, de modo que la nueva tupla que se intenta insertar hace referencia a una tupla inexistente de la tabla ALUMNO.

- `INSERTAR EN ALUMO-MATERIA (44835, 600, 8)`
  Esta operación es correcta. Lo que significa es que el alumno con la clave 44835 tiene de calificación 8 en la materia con clave 600. Se asume que existe una tupla para esa materia en una tabla de materias.

La operación de eliminación. La eliminación solo puede violar la integridad referencial en caso de que la tupla a eliminar esté referenciada por las claves foráneas de otras tuplas de la base de datos [8]. Considerando las tablas 2.1 y 2.2 se introducen algunos ejemplos ilustrativos:

- `ELIMINAR de la tabla ALUMNO la tupla con la clave 28000`
  Esta operación no es válida porque en la tabla ALUMNO-MATERIA existe una tupla cuya clave foránea (el número de cuenta) apunta a esta otra tupla.

- `ELIMINAR de la tabla ALUMNO-MATERIA la tupla con la clave 28000`
  Esta operación es válida y el resultado es la eliminación de una tupla.

Si una eliminación provoca una violación a la restricción de integridad referencial existen varias maneras en que puede ser manejado. Una de ellas consiste en rechazar la operación de borrado. Otra estrategia es propagar la eliminación a través de las tuplas que hacían referencia a la tupla que se intenta borrar. Una tercera posibilidad es alterar los valores de los atributos que referenciaban a la tupla que se intenta eliminar. Estos valores podrían ser nulos luego de la eliminación, o podrían apuntar a alguna tupla válida. Asignarles nulo sólo es posible si dichos atributos no forman parte de la clave primaria de la tabla en que se encuentran [8]. En general, cuando se especifica una restricción de integridad en el DDL, el DMBS permite especificar las opciones que se aplican en caso de que una operación de borrado genere una violación a dicha restricción [8].

La operación de actualización. La actualización se utiliza para cambiar los valores de uno o más atributos de una tupla o varias que existen en una tabla. Para seleccionar la información es necesario indicar una condición en los atributos de la tabla. Se consideran los siguientes ejemplos:

- `Actualizar la calificación del alumno de la tabla ALUMNO-MATERIA cuyo número de cuenta es 41440 a 9.`
  Esta operación es válida.
- `Actualizar el semestre del alumno de la tabla ALUMNO cuyo número de cuenta es 28000 a 3.`
  Esta operación es válida.

- `Actualizar el número de cuenta del alumno de la tabla ALUMNO-MATERIA cuyo número de cuenta es 41440 a 5000.`
  Esta operación es inválida porque viola la integridad referencial: no existe una tupla con el valor de 5000 en la tabla ALUMNO.

La actualización de un atributo que no forma parte de la clave principal ni de una clave foránea no suele introducir problemas. El DBMS solo debe verificar que el valor tiene el tipo de dato y dominio correctos.

2.3. Bases de datos no relacionales

Durante mucho tiempo se ha pensado que la tecnología basada en el modelo relacional basta para resolver prácticamente cualquier problema de administración y almacenamiento de datos. Esa línea de pensamiento se intensifica cuando propuestas alternativas como almacenes de documentos XML y bases de datos orientadas a objetos se absorben por sistemas manejadores de bases de datos relacionales (RDBMS). Lejos de reemplazar a los RDBMS, los complementan y extienden sus características funcionales [13].
Pero cuando las limitaciones de los RDBMS se hacen evidentes y los desafíos de almacenamiento de datos crecen, la idea de tener un solo sistema manejador de base de datos que resolviera todo ha sido cuestionada.

Uno de los sistemas pioneros de las bases de datos NoSQL es *BigTable,* el cual es una implementación del modelo basado en columnas, y es un componente de la infraestructura escalable de *Google* para procesamiento paralelo de grandes cantidades de datos. La infraestructura también está compuesta por un sistema de archivos distribuido, un sistema de coordinación distribuido y un ambiente de ejecución de MapReduce. La infraestructura da soporte al motor de ejecución de *Google* y a sus aplicaciones [35].
Otros antecedentes relevantes del enfoque no relacional ocurren en el año 2009. En esa época, el sitio de comercio electrónico *amazon.com* ajusta su arquitectura de software para hacerla más escalable. Esto implica el desarrollo de bases de datos no relacionales distribuidas y propietarias llamadas *Amazon S3* y *Dynamo* [14].
En ese mismo año se populariza el término *NoSQL* para referirse a bases de datos que no soportan el modelo relacional y no utilizan SQL. Dichos DMBS no relacionales están pensados para resolver problemas donde las bases de datos relacionales no son adecuadas, y agrupa a todas aquellas tecnologías que, por definición, lo que tienen en común es precisamente ser no relacionales [15]. Una definición más precisa de NoSQL son todas aquellas bases de datos de nueva generación que son no relacionales, distribuidas, de código abierto, sin esquema y horizontalmente escalables [17].

Empresas emergentes de la web fueron influenciadas por el DBMS columnar de *BigTable* y por el DMBS clave-valor de *Dynamo*; quienes desarrollaron sus propios sistemas de bases de datos no relacionales. El movimiento NoSQL también captó la atención de compañías como *Facebook*, quienes desarrollaron el sistema de *Cassandra*, usado después por *Twitter*. Asimismo, *Linkedln* desarrolló *Project Voldemort* y surgió *Ubuntu One*, un sistema de almacenamiento sincronizado en la nube basado en *CouchDB* [13].

Todas estas compañías tienen en común que ofrecen servicios web para una gran cantidad de usuarios y diariamente reciben trabajos pesados de lectura/escritura [5].
Adicionalmente a las aplicaciones web, las bases de datos NoSQL también soportan actividades diversas, incluyendo análisis predictivo y OLTP no crítico como transacciones entre organizaciones [36].

2.3.1. Bases de datos distribuidas

Un *sistema distribuido* es aquél que está compuesto por un conjunto de elementos de procesamiento interconectados mediante una red y que cooperan para llevar a cabo sus tareas asignadas. Un elemento de procesamiento es un dispositivo que puede ejecutar un programa por sí mismo [21].

Una *base de datos distribuida* es una colección de múltiples bases de datos interrelacionadas y conectadas sobre una red de computadoras.
Un sistema manejador de bases de datos distribuidas (DDBMS, *Distributed Database Management System*) es el software que permite la administración de la base de datos y hace la distribución transparente a los usuarios.
En ocasiones, un sistema de bases de datos distribuidas (*Distributed Database System*, DDBS) es referido como la base de datos y el DBMS que la administra [21]. Un DDBS no es una colección de archivos que pueden ser individualmente almacenados en un nodo de una red de computadoras. Para formar un DDBS, estos archivos no solamente están interrelacionados, sino que existe una estructura sobre ellos, y el acceso se hace mediante una interfaz común [21]. Todo DDBS tiene sus datos distribuidos físicamente a través de un conjunto de nodos que no necesariamente están geográficamente distantes. La comunicación de estos nodos es mediante una red en la cual cada nodo tiene su propio disco duro y su propia memoria principal [21].
Un sistema multi-procesador de memoria compartida no califica como un DDBS. Incluso si los procesadores no comparten elementos de hardware (memoria, disco duro, periféricos), un sistema multi-procesador es simétrico en el sentido de que sus componentes son idénticos; y son controlados por una o más copias del mismo sistema operativo que es responsable del control estricto de la asignación de tareas a cada procesador.
Esto no es verdad para un sistema distribuido, donde lo común es la heterogeneidad tanto a nivel de hardware como del sistema operativo. Las bases de datos que se ejecutan sobre sistemas multi-procesador se llaman *bases de datos paralelas* [21].
Un DDBS tampoco es un sistema interconectado por una red en donde la base de datos reside en un solo nodo y todas las solicitudes se envían a ese nodo. De ser el caso entonces los problemas de administración no serían diferentes de aquellos que se encuentran en una base de datos que opera en un entorno centralizado (figura 2.5). En realidad un DDBS es un sistema donde la base de datos se encuentra distribuida sobre un conjunto de nodos [21] (figura 2.6).

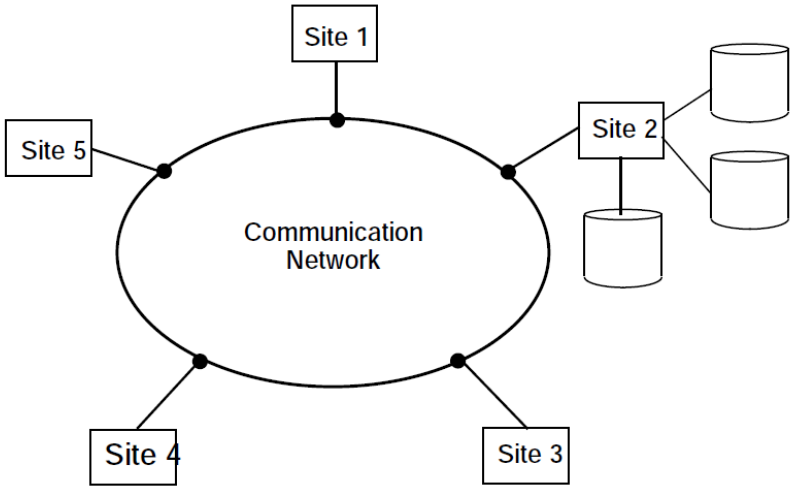

Figura 2.5. Base de datos centralizada en una red de computadoras [21].

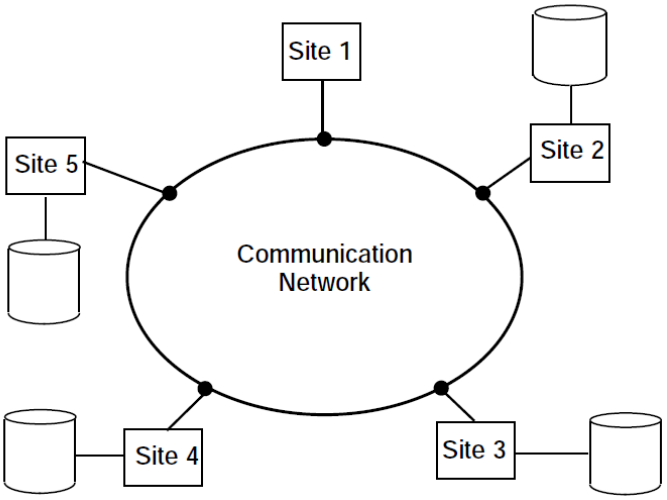

Figura 2.6. Entorno de un DDBS [21].

2.3.2. Limitaciones de las bases de datos relacionales

El conjunto de tecnologías NoSQL surge como respuesta a las limitaciones que presentan las bases de datos relacionales, de las cuales vale la pena mencionar las siguientes características:

1. **Escalabilidad**. Es común que el escalamiento de una base de datos relacional se logre *verticalmente,* agregando los elementos de hardware que sean necesarios para cubrir las necesidades (discos duros, memorias, microprocesadores, etc.). O bien se utiliza una computadora con el doble de procesamiento y almacenamiento. Pero esto es muy costoso. La alternativa de escalamiento horizontal es bastante más prometedora y económica, pero la tecnología de bases de datos relacionales no funciona adecuadamente si se distribuye en un conjunto de nodos interconectados. Esto es principalmente porque las bases de datos relacionales no están diseñadas para segmentar y replicar automáticamente sus datos, así que no es fácil distribuirlos [16].

2. **Complejidad**. Las bases de datos relacionales requieren que todos los datos se almacenen en tablas. No siempre es posible que los datos que se quieren introducir se ajusten al modelo de tablas. Cuando esto no ocurre, la estructura de la base de datos se puede volver muy compleja [13] [16].

3. **SQL es inflexible**. El uso de SQL es conveniente si los datos son por naturaleza estructurados. Cuando se necesita almacenar información de otro tipo, se vuelve difícil porque el lenguaje está diseñado para soportar solo datos estructurados que siguen el esquema rígido de tablas [16].

4. **Bloqueo**. Los RDBMS utilizan el bloqueo para proteger información que se va escribir durante una transacción [14]. Pero esto impacta negativamente en el rendimiento y no es imprescindible para todas las aplicaciones [13].
5. **Logging**. El manejo de los logs que se escriben a disco por cada transacción que ocurre introduce un costo en el rendimiento del sistema de bases de datos [13].

6. **Administración del buffer**. En muchos DBMS es ineficiente la manera en que los bloques del disco duro se guardan en memoria principal y en cómo se mapean registros de la base de datos al disco duro. [13].

7. **Costos de sincronización**. Cuando una base de datos relacional se encuentra distribuida; ésta suele introducir muchos costos de sincronización en el sistema al utilizar protocolos complejos [13].

2.3.3. Ventajas y desventajas de NoSQL

Las bases de datos no relacionales introducen los siguientes beneficios:

1. **Alto rendimiento**. Las velocidades de lectura y escritura en una base de datos NoSQL suelen ser mucho mayores que en un RDBMS tradicional. Por ejemplo, la tecnología de Hypertable es capaz de almacenar un billón de datos por día. Otro ejemplo es el software de Google BigTable, que puede procesar 20 petabytes de datos en un día [13]. En estudios comparativos se demuestra que el DBMS Cassandra escribe 2500 veces más rápido 50 GB de datos que MySQL [13].
   Algunos DBMS implementan las operaciones directamente en memoria principal y escriben al disco duro cada cierto tiempo, acelerando las operaciones de escritura.

2. **Escalamiento horizontal**. Si la base de datos experimenta cierto crecimiento, es posible agregar nodos a un sistema distribuido de tal forma que esto aporte procesamiento y almacenamiento de forma económica. Las bases de datos NoSQL tienen los mecanismos necesarios para la escalabilidad horizontal [13].

3. **Almacenamiento simple**. El modo de almacenamiento es mucho más simple que el esquema de tablas del modelo relacional. Esto afecta positivamente el rendimiento al no existir un esquema definido para guardar datos [13].

4. **Almacenamiento flexible**. Los esquemas de almacenamiento de las bases de datos no relaciones son mucho más flexibles que el esquema rígido de tablas. En realidad carecen de un esquema fijo. Cuando los datos a almacenar no se pueden traducir a tablas del modelo relacional entonces conviene buscar una solución por NoSQL.

5. **No ACID**. Sin un manejo estricto de las transacciones, el control de concurrencia no se hace por bloqueo sino que existen otros mecanismos. Como la durabilidad no es un elemento crítico en algunas aplicaciones web, no es necesario escribir logs por cada transacción. Ejemplos de datos no sensibles a la durabilidad incluyen copias de datos de sesión de usuarios y mensajes de texto en foros o redes sociales. [13].

A cambio de ofrecer los beneficios antes listados, las bases de datos NoSQL se enfrentan con los siguientes problemas [16]:

1. **Lenguaje de consultas no estandarizado.** En ausencia de un lenguaje estándar de consultas para bases de datos NoSQL, es necesario aprender cada lenguaje de consultas para un DBMS dado. Esto es cierto incluso dentro de cada categoría de almacenamiento NoSQL.

2. **Problemas en las transacciones**. Como consecuencia de no seguir ACID, existe un control más débil sobre la consistencia, la durabilidad y el aislamiento de las transacciones.

3. **Problemas de integridad**. Asegurar la integridad de los datos requiere de programación extra de forma manual y no es una función del DBMS. La integridad es entendida desde el punto de vista de las restricciones: de dominio, referencial de valor nulo, etc.

Entonces tanto los RDBMS como los no relacionales ofrecen beneficios pero también se enfrentan con ciertos conflictos y limitaciones. La decisión sobre qué tipo de DBMS conviene utilizar depende de un conjunto de factores que incluyen pero no se limitan a: el volumen de datos a almacenar, la concurrencia estimiada, la cantidad de operaciones que se hacen sobre la base de datos por unidad de tiempo, la escalabilidad deseada de la base de datos, el grado de integridad y consistencia que se desea, la naturaleza de los datos a almacenar y los tipos de operaciones más frecuentes que se desea hacer con los datos.

2.3.4. Teorema de CAP y propiedades BASE

En un documento titulado "*Towards Robust Distributed Systems*" del simposio de ACM PODC del año 2000, Eric Brewer presenta el teorema de CAP que ha sido adoptado por varias compañías de la Web y la comunidad NoSQL [13]. El acrónimo de CAP se refiere a lo siguiente:

C- Consistency (Consistencia). Se refiere a si un sistema se encuentra en un estado consistente luego de la ejecución de una operación. Un sistema distribuido se considera consistente si luego de una operación de actualización por un nodo, el resto de los nodos ven la actualización en un recurso de datos compartido [13].

A – Availabilty (Disponibilidad). Disponibilidad significa que un sistema está diseñado e implementado de tal forma que pueda continuar su operación si hay problemas de software o de hardware o falla un nodo [13].

P – Partition Tolerance (Tolerancia a la partición). Es la capacidad de un sistema para continuar su operación en la presencia de particiones de red. Esto ocurre si un conjunto de nodos en una red pierde conectividad con otros nodos. La tolerancia a la partición también se puede considerar como la capacidad de un sistema para permitir añadir y quitar nodos de forma dinámica [13].
El teorema dice que a lo más se pueden tener dos de estas tres características en un sistema de datos compartido.

Una base de datos NoSQL sigue el paradigma de sistemas BASE, en lugar del enfoque ACID del modelo relacional. BASE significa lo siguiente:

-**Ba**sically **A**vailable (Básicamente disponible)
-**S**oft state (Estado suave)
-**E**ventual Consistency (Consistencia eventual)

Las propiedades BASE se pueden resumir de la siguiente manera: una aplicación funciona básicamente todo el tiempo, no tiene que ser consistente todo el tiempo pero llegará eventualmente a un estado conocido [13].
De acuerdo con el teorema de CAP, el enfoque BASE se prefiere sobre ACID cuando un sistema necesita más disponibilidad y tolerancia a la partición que consistencia. Si por el contrario se necesita tolerancia a la partición y consistencia, entonces se requieren de las propiedades ACID [13]. Un ejemplo donde se prefiere BASE es en *Adobe ConnectNow,* que mantiene tres copias de la sesión de usuario, pero no requieren de revisiones de consistencia ni de persistencia.

2.3.5. Tipos de DBMS no relacionales

Existen cuatro categorías principales de DBMS no relacionales: almacenamiento clave-valor, bases de datos orientadas a columnas, bases de datos orientadas a documentos y bases de datos orientadas a grafos.

Almacenamiento clave-valor

Es un sistema que almacena valores indexados por claves. Estos sistemas pueden almacenar datos estructurados y no estructurados [16]. Ejemplos de DBMS de este tipo de almacenamiento incluyen Amazon Dynamo, Project Voldemort, RAMCloud y Flare [13].
Normalmente los valores se almacenan como arreglos de bytes, de tal forma que el contenido no es importante para la base de datos, solo la clave y el valor asociado. La posibilidad de almacenar cualquier tipo de valor se denomina *schema-less*, ya que no se necesita definir tablas ni columnas como en el modelo relacional [18].
Este tipo de almacenamiento es de alto rendimiento, muy escalable, muy flexible y de baja complejidad. La desventaja es que no permiten consultas complejas más allá de buscar por

su clave. Algunas implementaciones de este tipo de almacenamiento solo hacen operaciones sobre la memoria, por ejemplo: memcached, Oracle Coherence, JBoss Cache y WebSphere eXtreme Scale. Otras implementaciones utilizan el sistema de archivos. Es el caso de Amazon SimpleDB, VMWare Redis y Oracle BerkeleyDB [18].

Bases de datos orientadas a columnas

Como su nombre lo indica, guardan los datos en columnas en lugar de filas. Es decir, todos los atributos de una sola entidad de datos se almacenan de modo que se puede acceder a cada uno de ellos como una unidad [19]. Ejemplos de DBMS orientados a columnas incluyen a Cassandra, Hypertable y HBase [13].
La figura 2.7 ilustra cómo es el almacenamiento orientado a columnas comparado con el almacenamiento de filas.

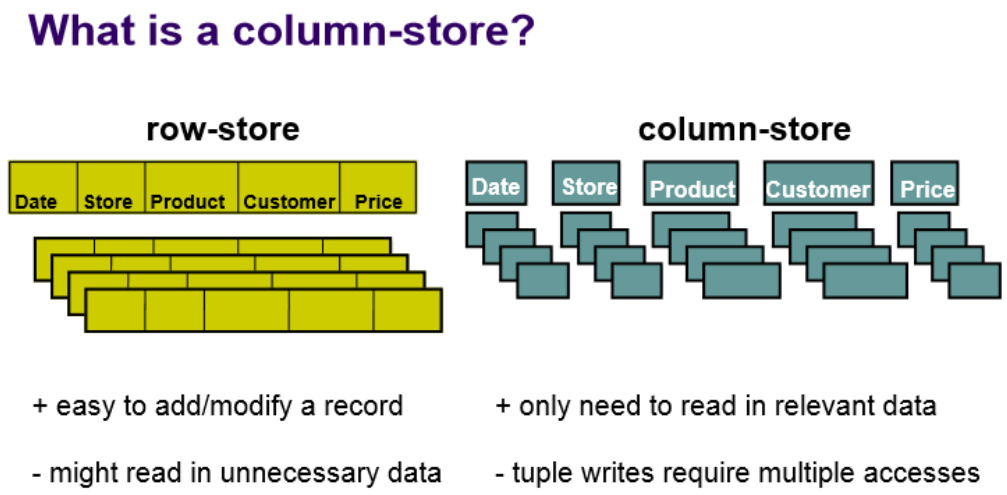

Figura 2.7. Comparación entre el almacenamiento de filas y el almacenamiento de columnas [20].

El almacenamiento orientado a columnas es especialmente eficiente cuando las lecturas de datos son masivas, y en cambio, se hacen pocas escrituras. Esto es porque en una consulta solo se obtienen los datos que interesan, no todos los de un registro, lo cual aumenta la eficiencia. El problema de las escrituras es que las columnas funcionan como unidades individuales y no son necesariamente contiguas, como en el almacenamiento por filas.
Las bases de datos orientadas a columnas se utilizan mucho en inteligencia de negocios [18].

En el DBMS de Cassandra, los bloques de construcción del almacenamiento orientado a columnas son las columnas, las súper columnas, familias de columnas y los *keyspace* [25].
Una columna es la unidad básica de este modelo de datos. Es una tupla que contiene el nombre de la columna, un valor y un timestamp. La figura 2.8 muestra esta estructura.

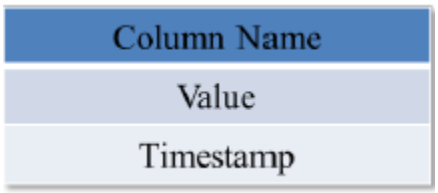

Figura 2.8. Una Columna en Cassandra [25].

Los timestamps almacenan el tiempo de última actualización de la columna y se usan para la resolución de conflictos. Un nombre de columna es análogo a un nombre de atributo en una tabla de una base de datos relacional [25].

Una súper columna está compuesta por un arreglo de varias columnas. Es especificada por un nombre y un mapa ordenado de columnas. Las columnas que pertenecen a una súper columna se agrupan usando un valor de búsqueda común, llamado *RowKey*. En otras palabras, una súper columna es un par clave-valor anidado de columnas. El par clave-valor externo forma la súper columna mientras que los pares internos corresponden a las columnas [25].
La figura 2.9 muestra la estructura de una súper columna.

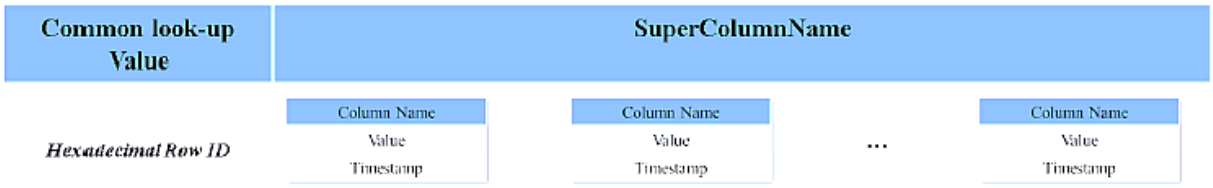

Figura 2.9. Una súper columna en Cassandra [25].

Una súper columna es análoga a un registro o tupla de una base de datos relacional.
Las súper columnas no tienen timestamps, a diferencia de las columnas [25].

Una familia de columnas contiene columnas o súper columnas agrupadas que utilizan un *RowKey* único y común. Puede verse como un conjunto de pares clave-valor, donde la clave es el *RowKey* y el valor es el mapa de nombres de columna.
La figura 2.10 presenta visualmente la estructura de una familia de columnas.

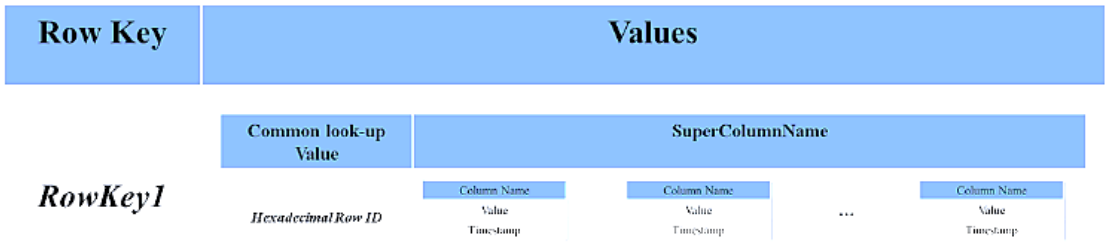

Figura 2.10. Una familia de columnas en Cassandra [25].

Las aplicaciones pueden definir familias de columnas y metadatos acerca de las columnas. Es una práctica común agrupar columnas relacionadas o que se acceden juntas en una familia de columnas [25].

Un *keyspace* es un contenedor que almacena datos que la aplicación utiliza. Los keyspaces pueden tener asociadas una o más familias de columnas, aunque no es siempre necesario que deban tener familias de columnas. Los keyspaces requieren que algunos atributos se definan, como nombres definidos por el usuario, estrategias de replicación y otros
Un keyspace es análogo a una base de datos en un RDBMS pero sin interrelaciones. [25].

Al igual que en el almacenamiento por clave-valor, las bases de datos orientadas a columnas son de alto rendimiento y muy escalables. Su complejidad de uso e implementación es baja y tienen flexibilidad moderada dado que no son apropiadas si se necesitan hacer muchas escrituras, o si se requiere acceder a datos de diferentes columnas de una misma entidad [13].

Bases de datos orientadas a documentos

Este tipo de bases de datos son en esencia un almacén clave-valor con la excepción de que el valor no se guarda sólo como un campo binario, sino con un formato definido de forma tal que el DBMS pueda entenderlo. Esto no quiere decir que tengan un esquema, siguen siendo libres de esquema y se usan solo dos campos. La diferencia es que el valor puede ser entendido por la base de datos [18]. Dicho formato a menudo es un documento JSON (*JavaScript Object Notation*), pero puede ser XML o cualquier otro. Si el DBMS interpreta los datos, puede hacer operaciones con ellos. De hecho, varias de las implementaciones de este tipo de bases de datos permiten consultas muy avanzadas sobre los datos, e incluso establecer relaciones entre ellos, aunque siguen sin permitir operaciones de reunión por cuestiones de rendimiento [18]. Ejemplos de DBMS orientados a documentos incluyen a CouchDB, MongoDB, Cloudkit y las bases de datos en XML.

En MongoDB, un documento es un conjunto de pares campo-valor en sintaxis de JSON.
La figura 2.11 presenta un ejemplo visual de un documento típico de MongoDB:

Figura 2.11. Un documento en MongoDB [2].

La ventaja de los documentos es que proveen de almacenamiento muy flexible. Además es posible envolver documentos y guardar arreglos, lo cual evita tener que ejecutar costosas operaciones de reunión [2]. Los documentos son análogos a las estructuras de los lenguajes de programación que asocian claves con valores, como tablas hash, diccionarios y mapas [2]. Formalmente, MongoDB almacena documentos en formato BSON (*Binary JSON*), una representación binaria de JSON [2]. Los documentos se almacenan en colecciones, que son

un conjunto de documentos relacionados que comparten índices. Las colecciones son análogas a las tablas de las bases de datos relacionales [2].
La figura 2.12 muestra conceptualmente una colección de MongoDB.

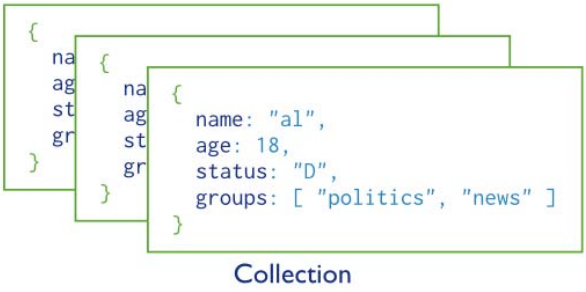

Figura 2.12. Una colección en MongoDB. [2]

El almacenamiento de documentos ofrece alto rendimiento, escalabilidad variable, alta flexibilidad y baja complejidad de implementación [13].

Bases de datos orientadas a grafos

Como su nombre lo indica, estas bases de datos almacenan los datos en forma de grafo.
Un grafo se representa como un conjunto de nodos (entidades) interconectados por aristas (relaciones). Esto permite darle importancia no solo a los datos, sino a las relaciones entre ellos. De hecho, las relaciones también pueden tener atributos y se pueden hacer consultas directas a relaciones, en vez de a los nodos. Además, al estar almacenadas de esta forma, es mucho más eficiente navegar entre relaciones que en un modelo relacional. Obviamente, este tipo de bases de datos sólo son aprovechables si la información se puede representar fácilmente como una red. Entre las implementaciones más usadas está Neo4J, Hyperbase-DB e InfoGrid [18]. El rendimiento y la escalabilidad de una base de datos orientada a grafos son variables, de alta flexibilidad y alta complejidad de implementación [13].

Comparación entre los distintos tipos de DBMS

La tabla 2.3 establece una comparación entre las distintas alternativas de almacenamiento de datos en base a rendimiento, escalabilidad, flexibilidad, complejidad y funcionalidad.
En [13] se establece que estas características son tendencias y generalidades de dichos sistemas, pero el rendimiento y la escalabilidad dependen mucho del tipo y frecuencia de operaciones que se aplican. Por ejemplo, los DBMS columnares no son apropiados si se requieren hacer muchas escrituras.

	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value Stores	high	high	high	none	variable (none)
Column stores	high	high	moderate	low	minimal
Document stores	high	variable (high)	high	low	variable (low)
Graph databases	variable	variable	high	high	graph theory
Relational databases	variable	variable	low	moderate	relational algebra

Tabla 2.3. Comparación entre los distintos tipos de DBMS [13].

Una segunda tabla comparativa ilustra más propiedades como el lenguaje de consultas, el protocolo de transmisión de datos, características de integridad, implementación de indexación, sistemas operativos soportados y modos soportados de distribución de datos. La tabla 2.4 establece dichas comparaciones [36].

Attributes		NoSQL Databases								
Database model		Document-Stored		Wide-Column Stored				Key-Value Stored		Graph-oriented
	Features	MongoDB	CouchDB	DynamoBD	HBase	Cassandra	Accumulo	Redis	Riak	Neo4j
Design & Features	Data storage	Volatile memory File System	Volatile memory File System	SSD	HDFS		Hadoop	Volatile memory File System	Bitcask LevelDB Volatile memory	File System Volatile memory
	Query language	Volatile memory File System	JavaScript Memcached-protocol	API calls	API calls REST XML Thrift	API calls CQL Thrift		API calls	HTTP JavaScript REST Erlang	API calls REST SparQL Cypher Tinkerpop Gremlin
	Protocol	Custom, binary (BSON)	HTTP, REST	-	HTTP/REST Thrift	Thrift & custom binary CQL3	Thrift	Telnet-like	HTTP, REST	HTTP/REST Tembedding in Java
	Conditional entry updates	Yes	Yes	Yes	Yes	No	Yes	No	No	
	MapReduce	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No
	Unicode	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	TTL for Entries	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	
	Compression	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	
Integrity	Integrity model	BASE	MVCC	ASID	Log Replication	BASE	MVCC	-	BASE	ASID
	Atomicity	Conditional	Yes	Yes	Yes	Yes	Conditional	Yes	No	Yes
	Consistency	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
	Isolation	No	Yes	Yes	No	No	-	Yes	Yes	Yes
	Durability (data storage)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-	Yes
	Transactions	No	No	No	Yes	No	Yes	Yes	No	Yes
	Referential integrity	No	No	No	No	No	No	Yes	No	Yes
	Revision control	No	Yes	Yes	Yes	No	Yes	No	Yes	No
Indexing	Secondary Indexes	Yes	Yes	No	Yes	Yes	Yes	-	Yes	-
	Composite keys	Yes	Yes	Yes	Yes	Yes	Yes	-	Yes	-
	Full text search	No	No	No	No	No	Yes	No	Yes	Yes
	Geospatial Indexes	Yes	No	No	No	No	Yes	-	-	Yes
	Graph support	No	No	No	No	No	Yes	No	Yes	Yes
Distribution	Horizontal scalable	Yes	Yes	Yes	Yes	Yes	Yes		Yes	No
	Replication	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes
	Replication mode	Master-Slave-Replica Replication	Master-Slave Replication	-	Master-Slave Replication	Master-Slave Replication	-	Master-Slave Replication	Multi-master replication	-
	Sharding	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
	Shared nothing architecture	Yes	Yes	Yes	Yes	Yes	-	-	Yes	-
System	Value size max.	16MB	20MB	64KB	2TB	2GB	1EB	-	64MB	
	Operating system	Cross-platform	Ubuntu Red Hat Windows Mac OS X	Cross-platform	Cross-platform	Cross-platform	NiX 32 entries Operating system	Linux *NIX Mac OS X Windows	Cross-platform	Cross-platform
	Programming language	C++	Erlang C++ C Python	Java	Java	Java	Java	C C++	Erlang	Java

Tabla 2.4. Comparación entre distintos DBMS NoSQL [36].

2.3.6. Escalamiento de una base de datos no relacional

Una base de datos puede escalar de tres maneras diferentes. Puede escalar en la cantidad de operaciones de lectura, el número de operaciones de escritura y el tamaño de la base de datos. Existen actualmente dos metodologías para lograr esto: replicación y sharding.

Replicación

La replicación significa que un elemento de datos está almacenado en más de un nodo. Esto es muy útil para incrementar el rendimiento de las lecturas de la base de datos, porque permite tener un balance de carga que distribuya las operaciones de lectura sobre muchas máquinas. También tiene la ventaja de que le da robustez a la base de datos contra fallas de nodos individuales. Si una máquina falla, entonces existe al menos otra con los mismos datos que puede reemplazar al nodo que falla [14].

La desventaja de la replicación son las operaciones de escritura. Una operación de escritura en una base de datos replicada tiene que hacerse en cada nodo que deba almacenar el respectivo elemento de datos. Una manera de hacer esto es que una operación de escritura se aplique directamente a todos los nodos. O bien la operación de escritura se hace en algunos nodos y después se envía asincrónicamente a todos los nodos. La elección de estas opciones decide las propiedades de consistencia y disponibilidad de la base de datos [14].

Sharding

En el sharding los datos de la base de datos se dividen y se distribuyen sobre varios nodos. La partición de los datos puede hacerse usando funciones hash que se apliquen a una clave primaria de elementos de datos para determinar la partición asociada. Esto implica que una tabla no se almacena en una sola máquina, sino en un clúster de nodos. Su ventaja es que los nodos pueden añadirse al clúster para incrementar la capacidad y el rendimiento de operaciones de escritura y lectura sin la necesidad de modificar la aplicación. Incluso es posible disminuir el clúster de la base de datos si la demanda disminuye [14].
La desventaja del sharding es que hace que algunas operaciones típicas en la base de datos se vuelvan muy complejas e ineficientes [14]. Si se usan más nodos en una base de datos particionada se hace más probable que alguna máquina o conexión de red falle. Por ello a veces se combina el sharding con replicación, lo cual hace el clúster más robusto contra de fallas de hardware [14].

2.3.7. Procesamiento de datos distribuido mediante MapReduce

MapReduce es un paradigma de programación popularizado por Google que se usa ampliamente para el procesamiento de una gran cantidad de datos en paralelo. Se basa en la programación de dos funciones: *Map* y *Reduce*. Los datos de entrada se dividen en pedazos, y diferentes nodos o procesadores ejecutan la función *Map* en cada pedazo. Otros procesadores, o incluso los mismos, aplican la función *Reduce* en las piezas de salida de la función *Map* [9].
Función MAP. Por lo regular las entradas del *Map* son pares clave-valor. La función *Map* se ejecuta por uno o más procesos, localizados en cualquier cantidad de nodos. Cada nodo

tiene una porción de los datos de entrada. La salida de la función *Map* es una lista de pares clave-valor [9]. Sin embargo, los tipos de claves y valores para salida de la función *Map* no necesariamente son los mismos que los tipos de entrada de las claves y valores. Las claves de la salida del *Map* no son claves (primarias, foráneas, etc.) en el sentido de las bases de datos relacionales. Esto es, pueden existir muchos pares con el mismo valor de clave [9].

El resultado de todos los procesos *Map* es una colección de pares clave-valor llamado *resultado intermedio*. Estos pares clave-valor son las salidas de la función *Map* aplicadas a todo par de entrada [9].

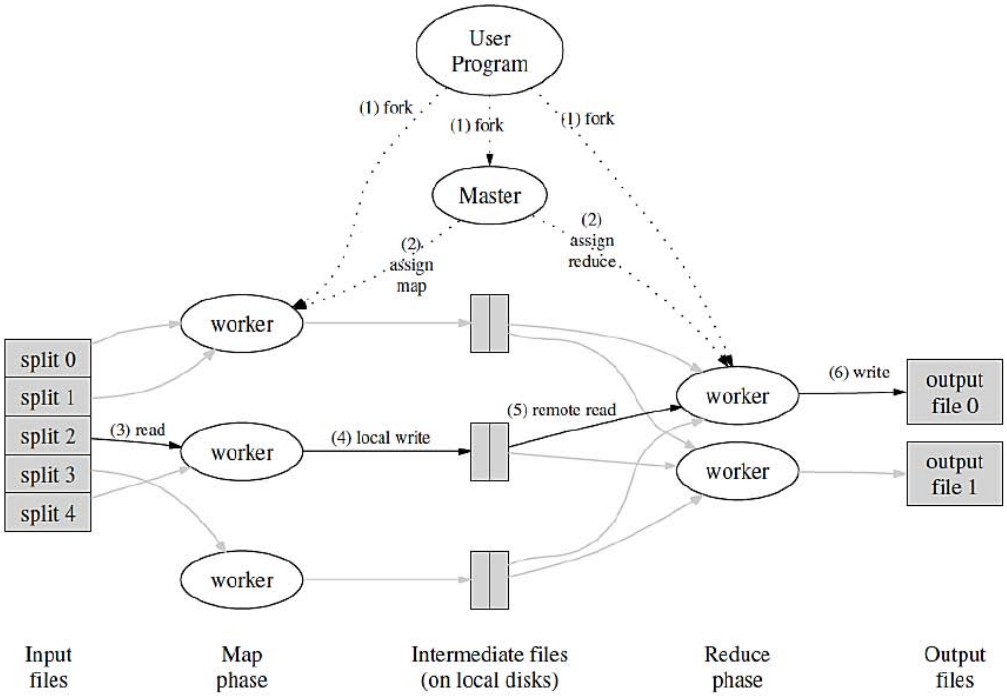

Figura 2.13. Ejecución de MapReduce [13].

Función REDUCE. La función *Reduce* también es ejecutada por un conjunto de procesos, localizados en cualquier cantidad de nodos. La entrada a la función *Reduce* es una sola clave del resultado intermedio, junto con la lista de todos los valores que aparecen para dicha clave del resultado intermedio. No se eliminan valores duplicados [9].
La función *Reduce* combina la lista de valores asociados con cada clave. En muchos casos, la operación *Reduce* es asociativa y conmutativa, y la lista entera de valores se reduce a un solo valor del mismo tipo que los de la lista de elementos. Por ejemplo, si la función hace una adición, entonces el resultado es una suma de la lista de los números [9].

El paradigma *MapReduce* ha sido adoptado por lenguajes de programación (Python), frameworks (Apache Hadoop), herramientas de Javascript (Dojo) y bases de datos NoSQL (Cassandra, MongoDB, CouchDB). En las bases de datos *MapReduce* se emplea para propósitos de análisis y tareas de preprocesamiento (en CouchDB sirve para generar vistas de datos) [13].

Al aplicar *MapReduce* a las bases de datos se envía el conjunto de claves a los nodos de almacenamiento, los cuales aplican de forma local la función *Map* a las claves. Los resultados intermedios se ordenan de forma consistente y luego son procesados por los nodos que aplican la función *Reduce* para producir resultados finales [13].
La figura 2.14 ilustra un ejemplo típico del proceso MapReduce.

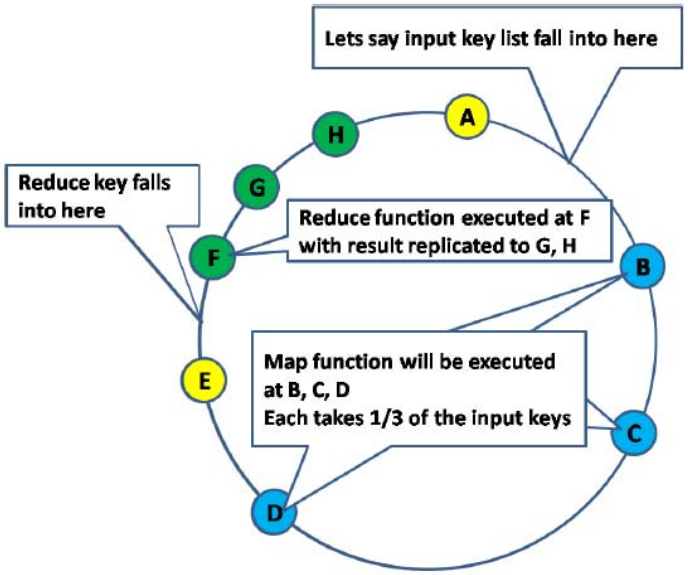

Figura 2.14. Aplicación de MapReduce a una base de datos [13].

2.4. Indexación

Muchas consultas hacen referencia a un pequeño conjunto de registros en un archivo. Por ejemplo, una consulta como "encontrar todas las cuentas de la sucursal Perryridge" o "encontrar el sueldo del número de cuenta A-101" referencian solo una pequeña fracción de los registros. Idealmente, el sistema debería localizar estos registros directamente. Para permitir estas formas de acceso, se diseñan estructuras adicionales que se asocian con archivos, las cuales se describen en las siguientes secciones [27].

2.4.1. Conceptos básicos

Un índice para un archivo en un sistema de base de datos funciona de la misma manera que el índice de un libro. Si se quiere revisar un tema en particular de un libro (especificado por una palabra o frase), se puede buscar el tema en el índice del libro, encontrar las páginas donde se encuentra y finalmente leer las páginas para obtener la información deseada [27].
El índice es mucho más pequeño que el libro y esto reduce el esfuerzo necesario para encontrar el tema que se está buscando [27].
Los índices en un sistema de base de datos juegan el mismo rol que los índices de libros. Por ejemplo, para obtener el registro de una cuenta dado el número de cuenta, el sistema de base de datos necesita buscar un índice para encontrar en qué bloque de disco reside el registro correspondiente, y luego trae el bloque de disco donde se encuentra el registro de la cuenta deseada [27].

Existen dos técnicas básicas de indexación [27]:

- **Índices ordenados:** Se basan en el almacenamiento ordenado de valores.

- **Índices hash:** Se basan en la distribución uniforme de valores a través de un rango de buckets. El bucket para el cual se asigna un valor se determina por una función, llamada función hash.

No hay una mejor técnica que otra, sino que cada técnica se ajusta mejor a aplicaciones de base de datos. Cada técnica debe ser evaluada en base a estos factores [27]:

- **Tipos de acceso:** Los tipos de acceso que son soportados eficientemente. Los tipos de acceso pueden incluir encontrar registros con valor específico de un atributo y encontrar registros cuyo rango de valores caen en un rango específico.

- **Tiempo de acceso:** El tiempo en que tarda encontrar un elemento particular de datos, o conjunto de elementos, usando la técnica en cuestión.

- **Tiempo de inserción:** El tiempo en que tarda insertar un nuevo elemento de datos. Este valor incluye el tiempo que tarda encontrar el lugar correcto para insertar el nuevo elemento de datos, así como el tiempo necesario para actualizar la estructura de índice.

- **Tiempo de eliminación:** El tiempo en que tarda eliminar un elemento de datos. Este tiempo incluye el tiempo que tarda encontrar el elemento de datos a ser borrado, así como el tiempo necesario para actualizar la estructura de índice.

- **Sobrecarga de espacio:** El espacio adicional ocupado por la estructura de índice. Si la cantidad de espacio adicional es moderada, usualmente vale la pena sacrificar el espacio para favorecer un mejor rendimiento.

El atributo o conjunto de atributos usado para buscar registros en un archivo se llama *clave de búsqueda*. Esta definición de clave difiere por completo de los conceptos de clave primaria, clave foránea, clave candidata y superclave [27].

2.4.2. Índices ordenados

Para obtener rápido acceso aleatorio a registros en un archivo, se utiliza una estructura de índice. Cada estructura de índice se asocia con una clave de búsqueda particular. Un índice ordenado almacena los valores de las claves de búsqueda en orden, y asocia con cada clave los registros que contiene [27].
Un archivo puede tener varios índices en distintas claves de búsqueda. Si el archivo se encuentra secuencialmente ordenado, un *índice primario* es un índice cuya clave de búsqueda también define el orden secuencial del archivo. A estos archivos comúnmente se les llama *archivos de índice secuencial* y representan uno de los esquemas de indexación más antiguos

en los sistemas de bases de datos. Se utilizan en aplicaciones que requieren procesamiento secuencial de todo el archivo y para acceso aleatorio a registros individuales [27].

La figura 2.12 muestra un archivo de registros ordenado secuencialmente que almacena información de cuentas bancarias, en donde la clave de búsqueda es el nombre de la sucursal.

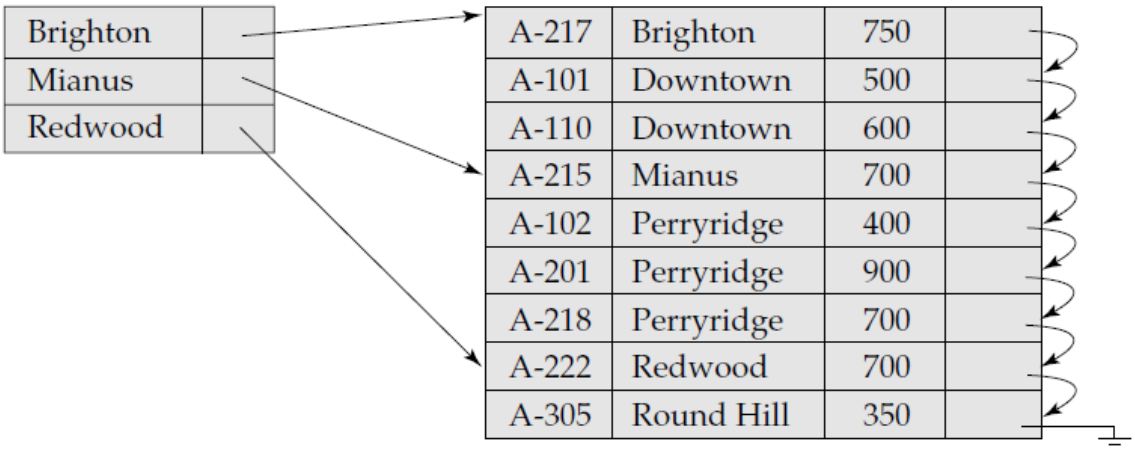

Figura 2.15. Archivo secuencial para registros de cuentas [27].

Un registro índice o entrada índice consiste del valor de la clave de búsqueda, y apunta a uno o más registros con ese valor de la clave de búsqueda. El apuntador a un registro consiste del identificador del bloque de disco y un valor dentro del bloque de disco que identifica el registro dentro del bloque [27].
La desventaja principal de la indexación secuencial de archivos es que el rendimiento se degrada conforme el archivo crece, tanto para búsquedas por índice como para procesamiento secuencial a través de los datos. Aunque esta degradación puede remediarse reorganizando el archivo, las reorganizaciones frecuentes son indeseables [27].

2.4.3. Índices de archivos mediante árboles B+-.

La estructura de indexación árbol B+- es la más usada de las estructuras de indexación para mantener la eficiencia independientemente de inserciones y eliminaciones de datos.
Un árbol B+- toma la forma de un árbol balanceado en el cual todo camino desde la raíz hasta una hoja del árbol es de la misma longitud. Cada nodo no hoja del árbol tiene entre $n/2$ y n nodos hijos, donde n es fijo para un árbol particular [27].
Una estructura B+- implica costos de rendimiento en inserciones y eliminaciones, además de costos de almacenamiento. Estos costos adicionales son aceptables incluso si los archivos son frecuentemente modificados, porque se evita el costo de reorganización [27].
Un nodo de un árbol B+- contiene hasta $n-1$ claves de búsqueda K_1, K_2, … K_{n-1} y n apuntadores P_1, P_2, … P_n. Las claves de búsqueda dentro de un nodo se guardan ordenadas, de modo que si $i < j$, entonces $K_i < K_j$ [27].
La figura 2.13 muestra la estructura común de un nodo en un árbol B+-.

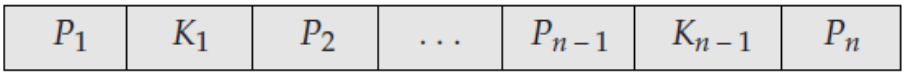

Figura 2.16. Estructura de un nodo en un árbol B+- [27].

El apuntador P_i para *i = 1, 2,...., n-1* de un nodo hoja apunta directamente a un archivo de registro con clave de búsqueda K_i.
La figura 2.14 muestra un ejemplo de la estructura de un nodo hoja donde n = 3 y la clave de búsqueda es el nombre de la sucursal bancaria.

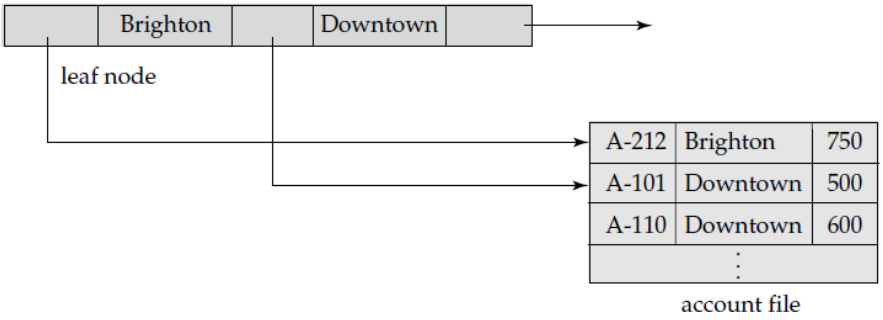

Figura 2.17. Un nodo hoja para el índice de cuenta en un árbol B+- [27].

Cada hoja puede tener hasta *n – 1* valores. Los rangos de valores en cada hoja no se traslapan. Entonces, si L_i y L_j son nodos hoja y i < j, entonces toda clave de búsqueda en L_i es menor que toda clave de búsqueda en L_j [27]. Como existe un orden lineal en las hojas basado en los valores de la clave de búsqueda que contienen, se utiliza el apuntador P_n para encadenar juntos los nodos hoja en el orden de clave de búsqueda. Este ordenamiento permite el procesamiento secuencial eficiente del archivo [27].
La estructura de los nodos no hoja es la misma que la de los nodos hoja, excepto que todos los apuntadores refieren a nodos del árbol. Un nodo no hoja puede tener *n* apuntadores, y debe tener al menos *n/2* apuntadores [27].
Un nodo con *m* apuntadores con un apuntador P_i para *i = 2, 3,..., m – 1,* P_i apunta al subárbol que contiene la clave de búsqueda menor que K_i y mayor o igual que K_{i-1}.
El apuntador P_m apunta a la parte del subárbol que contiene aquellas claves mayores o iguales que K_{m-1}., y el apuntador P_1 apunta a la parte del subárbol que contiene aquellas claves menores que K_1 [27].
A diferencia de otros nodos, el nodo raíz puede tener menos que *n/2* apuntadores; sin embargo, debe tener al menos dos apuntadores, a menos de que el árbol consista de un solo nodo. Siempre es posible construir un árbol B+- para cualquier *n* y satisfacer los anteriores requerimientos [27].

La figura 2.15 muestra un árbol B+- para el archivo de cuentas con n = 5. Por simplicidad, se omiten los apuntadores al archivo y los apuntadores nulos.

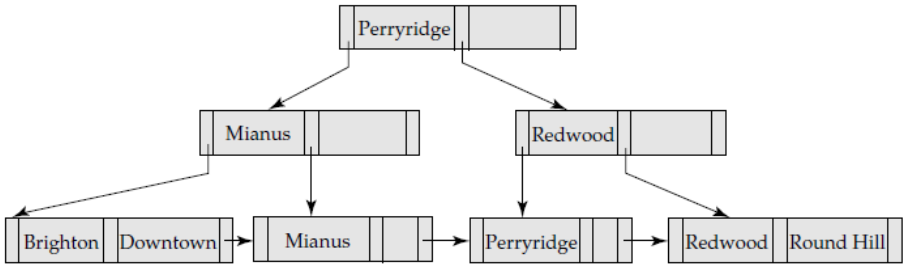

Figura 2.18. Árbol B+-para el archivo de cuentas [27].

La longitud de este árbol de la raíz a un nodo hoja es la misma. Esta propiedad es un requerimiento para un árbol B+-. De hecho, la "B" en B+- es por "balanceado".
Es la propiedad de balance de los árboles B+- lo que asegura un buen rendimiento para búsqueda, inserción y eliminación [27].

2.4.4. Índices en MongoDB

Los índices en MongoDB permiten el escaneo rápido de documentos, evitando el recorrido de toda la colección y utilizando en su lugar estructuras de datos especiales que almacenan sólo una porción de ésta, mediante árboles B. El índice almacena el valor de campos específicos, ordenados por el valor de dichos campos [28]. Los índices se definen en el nivel de colección y si son apropiados, MongoDB puede utilizarlos para limitar el número de documentos que debe inspeccionar [28].
Cuando los criterios de búsqueda y la proyección de la consulta incluyen sólo los campos indexados, MongoDB puede devolver resultados directamente desde el índice sin escanear ningún otro documento [28].
La figura 2.16 muestra un ejemplo de un índice y una consulta que lo utiliza.

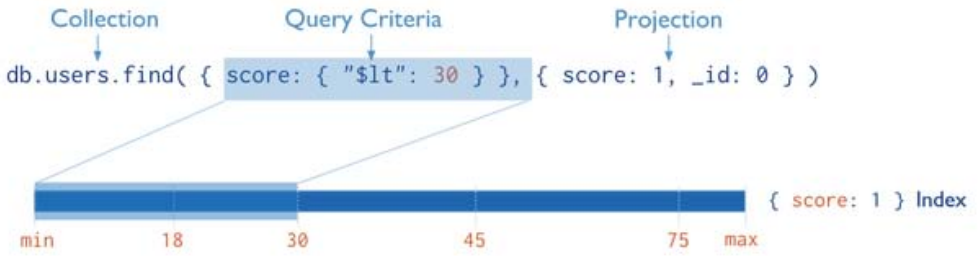

Figura 2.19. Diagrama de una consulta que utiliza un índice. La búsqueda se reduce al rango de documentos con campo *score* menor a 30 [28].

En el ejemplo, el índice se crea sobre el campo *score* y almacena los documentos en orden ascendente. Como los criterios de búsqueda y de proyección se hacen sólo sobre ese campo, entonces MongoDB puede usar el índice [28]. El criterio de búsqueda "$lt:30" significa *less than 30* o menor que 30. Una proyección en MongoDB utiliza el valor 1 para campos que se desean recuperar y un 0 para campos que se desean excluir. Por lo regular solo se especifica qué campos se quieren recuperar, pero MongoDB por default siempre regresa el campo *_id* que funciona como clave primaria, así que para omitirlo hay que indicarlo explícitamente.

Los tipos de índices que soporta MongoDB son:

- Sencillos
- Compuestos
- Multiclave
- Geoespaciales
- De texto
- Hash

El índice de la figura 2.16 es sencillo pues se utiliza un solo campo. Los índices compuestos mantienen referencias a múltiples campos dentro de una colección de documentos [29]. La figura 2.17 ilustra un ejemplo de un índice compuesto por dos campos.

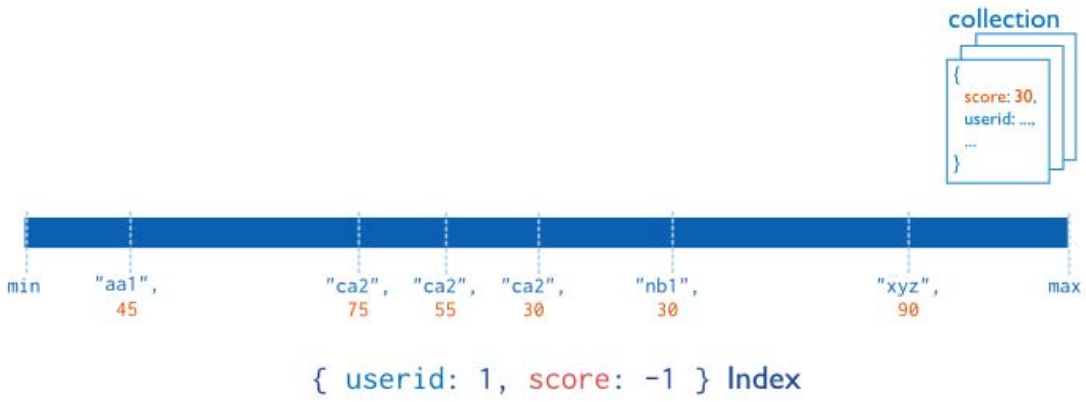

Figura 2.20. Diagrama de un índice compuesto por los campos *userid* (ascendente) y *score* (descendente) [29].

En la figura 2.17, el índice primero ordena ascendentemente en base al campo *userid* y después descendentemente por el campo *score* [29].
Los índices compuestos soportan consultas en cualquier prefijo de los campos indexados. Los prefijos de índice son el subconjunto de inicio de los campos indexados.
Por ejemplo, dado el siguiente índice:

```
{"item": 1, "location": 1, "stock": 1}
```

MongoDB puede usar este índice para soportar consultas que incluyen:

- El campo *item*
- El campo *item* y el campo *location*
- El campo *item* y el campo *location* y el campo *stock*
- El campo *item* y el campo *stock*; sin embargo, este índice puede ser menos eficiente que un índice solo en *item* y *stock*.

Cualquier otra consulta no es soportada por este índice [29].
La sintaxis para crear un índice en el shell de mongo es la siguiente:

```
db.collection.ensureIndex ({campo 1: 1|-1, campo 2: 1|-1,…. campo n: 1|-1})
```

Donde *collection* es el nombre de la colección.
Si se asigna el valor de 1 los valores del campo se ordenan ascendentemente y si el valor es -1 el ordenamiento es descendente.

2.5. Resumen

Este capítulo introduce conceptos fundamentales de la teoría de las bases de datos, así como características principales del modelo relacional, incluyendo las operaciones, restricciones y estructuras de datos que utiliza.
Se hace énfasis en las bases de datos no relacionales, pues constituyen el objeto de estudio de esta tesis. Por ello se cubre un conjunto de temas relevantes como los contrastes de NoSQL con el modelo relacional, las ventajas y desventajas de NoSQL, los tipos de DBMS no relacionales que existen y el teorema de CAP. Dichos temas son comunes al conjunto de tecnologías NoSQL, los cuales son DBMS gratuitos, libres de esquema y de naturaleza distribuida.
Finalmente se revisan algunos conceptos de la teoría de indexación, la cual juega un papel importante en esta investigación.

Capítulo 3. Trabajos Relacionados

A continuación se abordan tres investigaciones relacionadas a esta tesis, las cuales trabajan con tecnología NoSQL. En primer lugar se hace referencia a la tesis de maestría de Harsha Raja, titulada *Referential Integrity in Cloud NoSQL Databases* (Integridad referencial en bases de datos NoSQL en la nube) [24], publicada en 2012, donde propone un API de control de integridad referencial para el DBMS de Cassandra. En segundo lugar se resume el artículo de Sanobar Khan y Vanita Mane, *SQL Support over MongoDB using Metadata* (Soporte de SQL sobre MongoDB usando metadatos) [25]. En ese estudio se hace una comparación entre el DBMS no relacional MongoDB y el RDBMS MySQL y se propone una arquitectura de software que integra operaciones SQL sobre MongoDB. El tercer trabajo relacionado es de autoría de Kalin Georgiev, titulado *Referential integrity and dependencies between documents in a document oriented database* (Integridad referencial y dependencias entre documentos en una base de datos orientada a documentos) [26], donde propone el control de integridad referencial desde un enfoque de verificación y utilizando MapReduce.

3.1. Integridad referencial en bases de datos NoSQL en la nube

La investigación hecha en [24] consiste en el desarrollo de una interfaz de programa de aplicación (*Application Programming Interface*, API), la cual funciona como una capa intermedia entre una base de datos no relacional y aplicaciones, con el propósito de mantener la integridad referencial. La API provee de las operaciones necesarias de creación, lectura, actualización y eliminación de datos asegurando que se conservan las restricciones de integridad referencial. Dichas restricciones son metadatos y se proveen de cuatro enfoques diferentes para almacenarlos. El rendimiento de dichos enfoques es medido usando diferentes restricciones de integridad referencial y los experimentos se hicieron sobre el DBMS de Cassandra, uno de los sistemas más comunes y prominentes de NoSQL [24].

Cassandra es un sistema que pertenece a la categoría de DBMS de almacenamiento por columnas de acuerdo a ciertas taxonomías de NoSQL [13]. Para la investigación de [24] este almacenamiento es en realidad un subconjunto de las bases de datos clave-valor.

De cualquier modo, lo que es importante es que los metadatos que Cassandra almacena son de sólo lectura y dicha información tiene que ver con cómo están distribuidos los datos en un clúster, pero no tiene información alguna sobre restricciones de integridad, como ocurre con los RDBMS [24].

De las cuatro soluciones propuestas para el control de la integridad referencial, dos de ellas son por metadatos embebidos, lo que implica que los metadatos se guardan junto con los datos de la base de datos. Las otras dos soluciones separan los metadatos de los datos almacenados en la base de datos [24]. Los metadatos contienen información necesaria para conservar la integridad referencial, incluyendo datos sobre claves primarias, claves foráneas, columnas referenciadas y referenciantes, restricciones, etc. [24]

Los metadatos están especificados por los siguientes elementos [24]:

- **Nombre de restricción (ConstraintName):** Nombre utilizado para identificar alguna restricción de clave primaria o foránea en los metadatos.

- **Keyspace**: Representa el nombre del keyspace al que pertenece la restricción.

- **Tipo de restricción (ConstraintType):** Denota el tipo de restricción. Los posibles valores son **P, R** y **F. P** corresponde a una restricción de clave primaria, mientras que **R** representa una restricción de integridad referencial que una entidad tiene sobre otra. Finalmente **F** representa dependencias existentes de otra entidad.

- **Familia de columnas (ColumnFamily):** Se refiere a la familia de columnas a la cual la restricción aplica.

- **RKeyspace**: Es el nombre del keyspace en el cual la restricción se aplica.

- **RConstraintName:** Representa la restricción referenciada. Para la restricción **R**, esto representa la restricción de clave primaria referenciada, y para la restricción **F**, muestra las dependencias de unas entidades respecto de otras.

- **RColumn**: Indica la columna de la clave primaria sobre la cual la restricción se aplica. Para restricciones de clave primaria, esto guarda el nombre de la columna de la clave primaria. Para restricciones de clave foránea, este campo denota la columna referenciada.

- **Regla de eliminación (DeleteRule):** Almacena el tipo de regla de manipulación de datos de la restricción para una operación de eliminación. Por motivos de simplicidad, esta regla también aplica para operaciones de actualización. Las posibles reglas son borrar en cascada o no borrar.

Un ejemplo del almacenamiento de metadatos se provee en la figura 3.1, que almacena información de una universidad.

Constraint Name	Keyspace	Constraint Type	Column Family	RKeyspace	RConstraint Name	RColumn	DeleteRule
CONST100	University	P	Student	University		StudentId	
CONST200	University	P	Course	University		CourseId	
CONST300	University	P	Enrolment	University		RowId	
CONST400	University	R	Enrolment	University	CONST100	StudentId	CASCADE
CONST500	University	R	Enrolment	University	CONST200	CourseId	NODELETE
CONST600	University	F	Course	University	CONST500	CourseId	NODELETE
CONST700	University	F	Student	University	CONST400	StudentId	CASCADE

Figura 3.1. Ejemplo del almacenamiento de metadatos [24].

3.1.1. Solución 1. Metadatos en súper columnas

En esta solución los metadatos se encuentran embebidos junto con los datos de la base de datos almacenándolos dentro de cada súper columna. Esto es, cada súper columna o familia de columnas almacena sus metadatos en una columna especial llamada METADATOS. Como los metadatos son comunes al keyspace, todas las súper columnas en cada familia de columnas contienen el mismo valor en la columna de METADATOS [24].

Los metadatos contienen todos los elementos antes descritos y cada súper columna almacena todas las restricciones que pertenecen al keyspace de la columna METADATOS. Como todas las restricciones se almacenan juntas, es posible obtener todas las restricciones relevantes de una súper columna de la columna METADATOS [24].
Visualmente esto equivale al ejemplo de la figura 3.2.

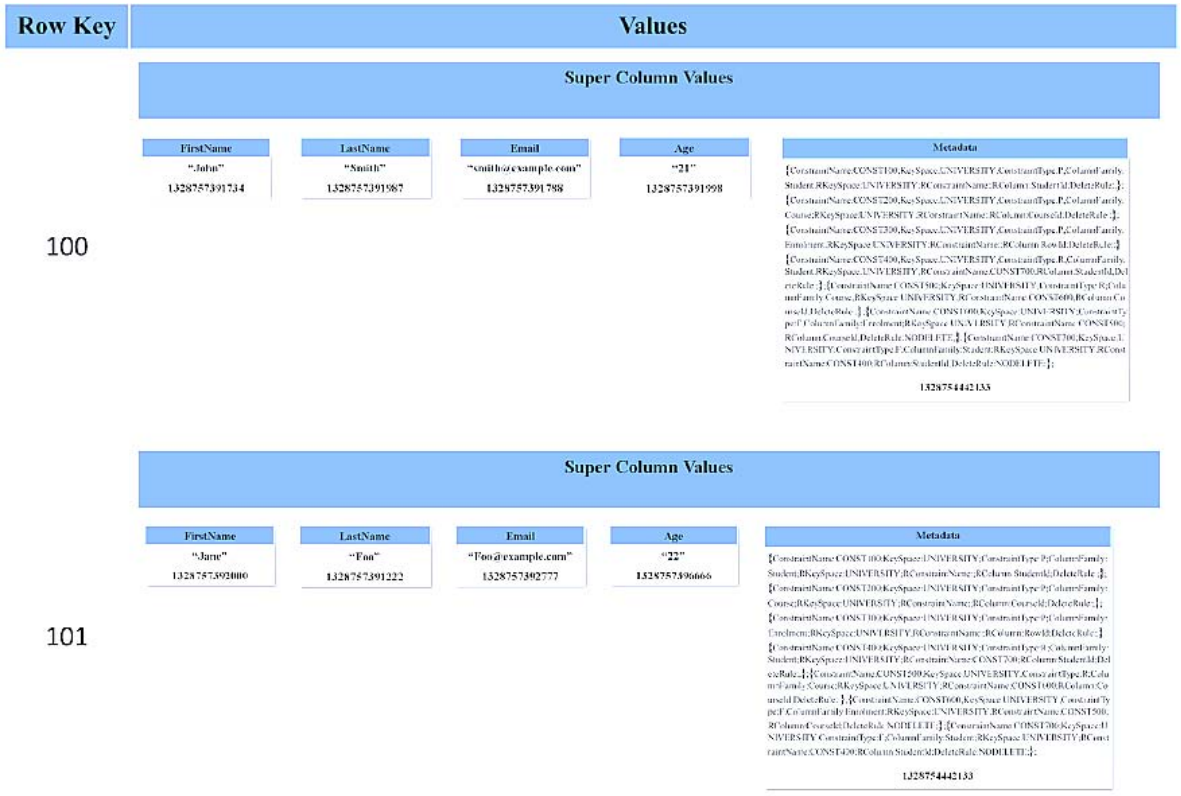

Figura 3.2. Almacenamiento de metadatos en súper columnas [24].

Algunos caracteres especiales se utilizan para marcar y delimitar restricciones. Los metadatos en esta solución contienen todas las restricciones referentes a un keyspace [24].

3.1.2. Solución 2. Metadatos como un registro superior

En esta solución los metadatos se encuentran embebidos con los datos y existen dentro de la misma familia de columnas, lo cual es similar a la solución 1. Lo que es diferente es que los metadatos se almacenan solo una vez en la familia de columnas como un registro superior.

Específicamente, los metadatos se almacenan en la primera súper columna en una familia de columnas. Este registro superior sólo contiene la columna de METADATOS que tiene como valor la información de los metadatos mientras que el resto de las súper columnas de la familia de columnas tienen diferentes columnas que almacenan los datos de la base de datos. El registro superior es común para todas las súper columnas.
La figura 3.3 representa un ejemplo de aplicación de esta solución [24].

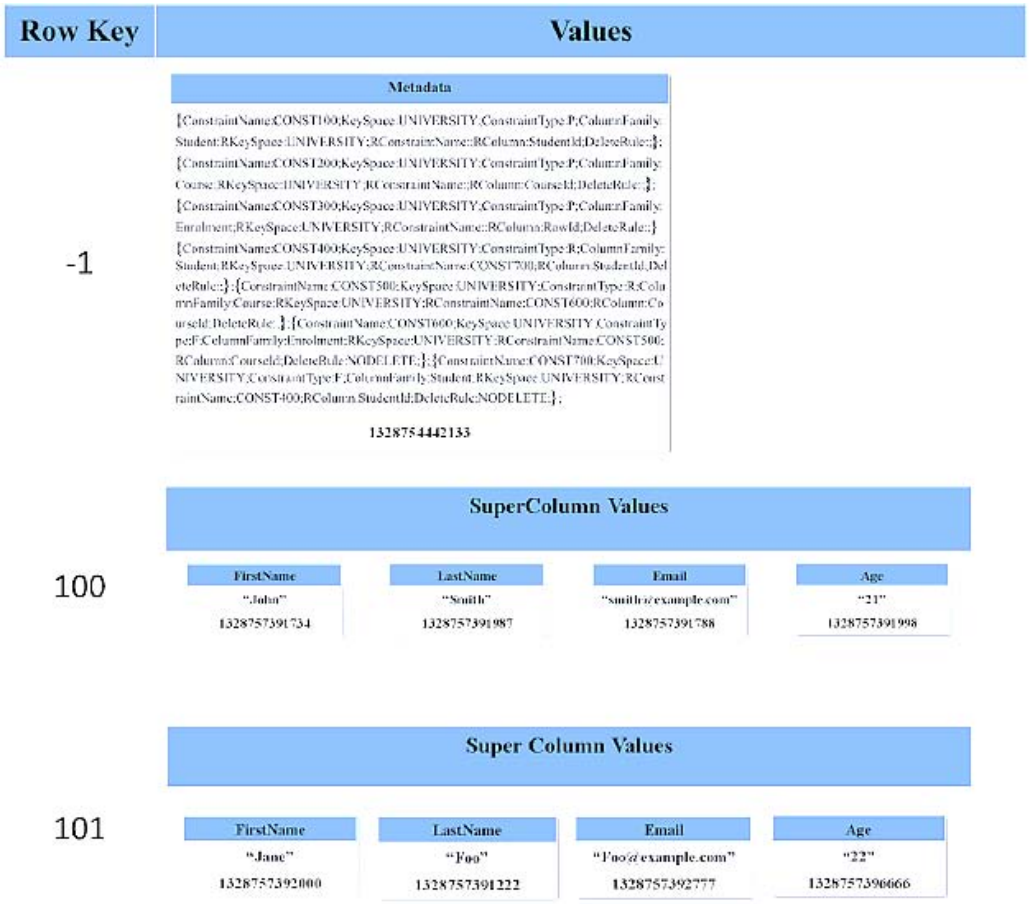

Figura 3.3. Almacenamiento de metadatos en un registro superior [24].

En esta solución, los metadatos para cada familia de columnas contienen todas las restricciones que pertenecen al keyspace. Como en la solución 1, existen caracteres especiales que marcan y delimitan las restricciones [24].
Esta solución reduce la redundancia de los metadatos de la solución 1, donde los metadatos se almacenan en cada súper columna de una familia de columnas y se replican a través del clúster junto con la familia de columnas. La solución 2 reduce esta redundancia y centraliza los metadatos en un registro superior dentro de la familia de columnas. Entonces se necesita menos almacenamiento para guardar los metadatos que en la solución 1 [24].

3.1.3. Solución 3. Familia de columnas de metadatos

En la solución 3, los metadatos para todas las familias de columnas en un keyspace se almacenan como una familia de columnas separada llamada METADATOS. En esta aproximación, los metadatos se separan de los datos de la base de datos y se almacenan de forma centralizada donde todas las restricciones de clave primaria y de clave foránea de todas las familias de columnas dentro de un keyspace se guardan en un solo lugar. Las otras familias de columnas almacenan sólo los datos de la base de datos y no contienen ninguna información de metadatos. En este enfoque, todas las restricciones se guardan en súper columnas en la familia de columnas de METADATOS [24].
Un ejemplo de esta solución se encuentra en la figura 3.4.

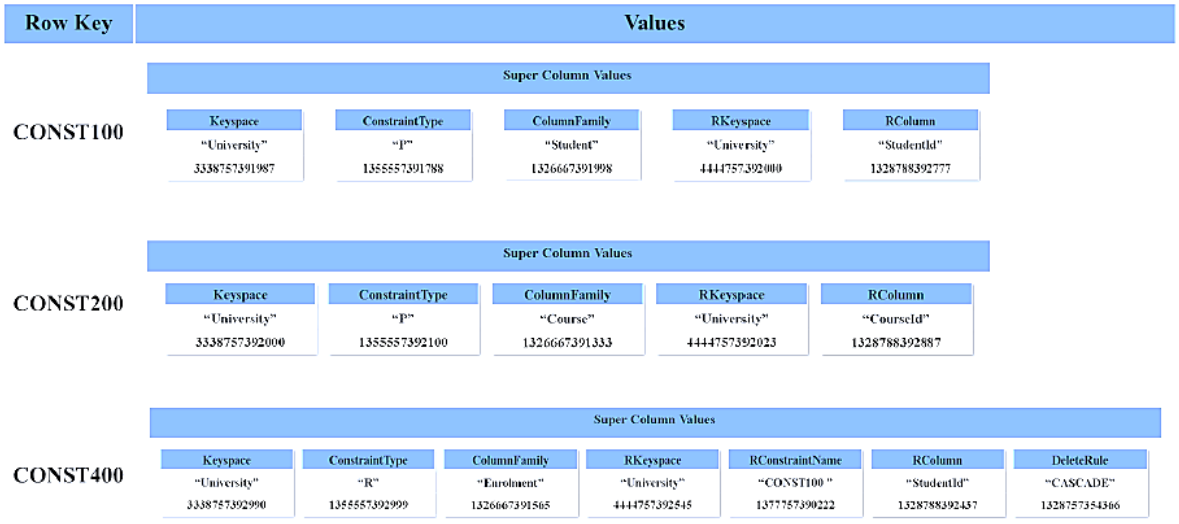

Figura 3.4. Almacenamiento de metadatos en familias de columnas [24].

Las diferentes partes de las restricciones se guardan como columnas separadas en la familia de columnas METADATOS. Entonces no se necesitan caracteres especiales para identificar las restricciones [24].

Cuando se hace una operación sobre una familia de columnas del keyspace, se generan triggers de validación de integridad referencial. Para lograr esto, es necesario acceder primero a la familia de columnas de metadatos para obtener las restricciones relevantes para la familia de columnas donde se hizo la operación [24].

Esta aproximación es similar a como los RDBMS almacenan información sobre dependencias, donde los metadatos contienen información sobre tablas, sus dependencias y muchas otras propiedades. Tales metadatos se almacenan en tablas del sistema, que están separadas de los datos que almacena la base de datos [24].
Este diseño supera los retos de las soluciones 1 y 2, donde las columnas de metadatos pueden ser muy grandes y por tanto difíciles de mantener. Otro problema es que los metadatos se tienen que cambiar en todo lugar donde están repetidos, en caso de que se tengan que hacer alteraciones a las restricciones [24].

3.1.4. Solución 4. Clúster de metadatos

En la solución 4, los metadatos se almacenan en una familia de columnas separada de los datos de la base de datos, como en la solución 3. Sin embargo, en esta solución, la columna de metadatos se almacena en un clúster separado de Cassandra en lugar de en el mismo clúster. Como dicho clúster no requiere de muchos nodos, los metadatos no se replican tanto como en las soluciones previas pues la replicación de metadatos es sólo dentro del clúster. La figura 3.5 ilustra gráficamente esta solución [24].

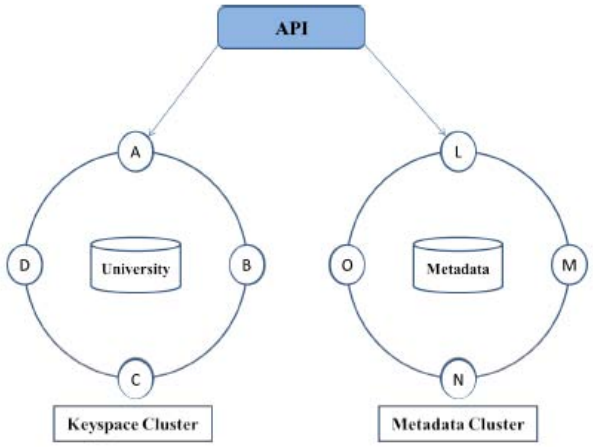

Figura 3.5. Almacenamiento de metadatos en un clúster separado del keyspace [24].

Para hacer operaciones sobre los datos, es necesario conectarse a ambos clústeres: el que contiene el keyspace y el que contiene los metadatos. Cuando una operación de manipulación de datos se aplica a una familia de columnas, los metadatos relevantes deben ser recuperados del clúster de metadatos. Sin embargo, como esta conexión requiere de cierto tiempo, se implementa una caché para reusar metadatos recientes en futuras operaciones sobre esa familia de columnas. Las ventajas del caché es que si el clúster de metadatos se encuentra inactivo o no responde, los metadatos aún se pueden recuperar de la caché para seguir haciendo operaciones de validación de datos. Esto también ahorra tiempo ya que no es necesario conectarse a la familia de columnas de metadatos cada vez que se accede a los metadatos [24]. Esta aproximación se inspira en la manera en que muchos sistemas distribuidos guardan metadatos en clústeres de servidor de metadatos [24].

3.1.5. Implementación de las restricciones de integridad en un DBMS NoSQL

Las cuatro soluciones propuestas en [24] almacenan las restricciones de integridad referencial como familias de columnas de metadatos, donde cada solución las almacena de una forma particular. Tales metadatos se acceden cuando se hacen operaciones de creación, lectura, actualización y eliminación sobre las familias de columnas. Las validaciones requieren acceder a las restricciones de clave foránea de una familia de columnas y sus asociadas restricciones de clave primaria. Éstas se procesan en orden para obtener los valores contenidos en sus restricciones [24].
Por cada solución se utilizan métodos específicos para obtener y procesar metadatos, y estos métodos junto con las soluciones se incorporan en una sola API.

La API experimental implementa el diseño de las soluciones y provee manejadores para ejecutar operaciones de manipulación de datos y validar la integridad referencial [24].
La API experimental fue diseñada de tal forma que pueda ser usada por las aplicaciones para mantener dependencias dentro de sus respectivos keyspaces independientemente de los esquemas de keyspace o las estructuras de una familia de columnas. Las aplicaciones deben proveer todas las restricciones de integridad referencial, ya que éstas no se deducen automáticamente. La API valida la integridad referencial basada en los metadatos que provee la aplicación y sus familias de columnas [24].

3.2. Soporte de SQL sobre MongoDB usando metadatos

La propuesta de [25] integra en un sistema, consultas del lenguaje SQL sobre una base de datos NoSQL mediante una capa de software intermedia llamada *metadatos*. En este contexto, metadatos se refiere a un conjunto de paquetes de software que actúan como interfaz entre una aplicación de JAVA y una base de datos MongoDB. No son los metadatos que especifican restricciones de integridad en la investigación de [24].

3.2.1. Generalidades de MongoDB

MongoDB es un DBMS orientado a documentos y sin esquema. El sistema es escalable y está escrito en lenguaje C++. El concepto de fila o tupla de las bases de datos relacionales es reemplazado por el de *documento* [25].

Características. MongoDB soporta estructuras de datos BSON (JSON codificado en binario) para almacenar tipos de datos complejos; tiene un lenguaje de consultas poderoso; provee rápido acceso a los datos; almacena y distribuye archivos binarios como imágenes y videos; soporta un protocolo de fácil uso para almacenar grandes archivos y metadatos de archivos; utiliza archivos mapeados en memoria para tener alto rendimiento [25].

Modelo de datos. MongoDB mantiene un conjunto de colecciones. Una colección no tiene un esquema predefinido como las tablas del modelo relacional, y almacena datos en documentos. Se usa el formato BSON para almacenar documentos [25].
Un documento es un conjunto de campos interrelacionados y puede ser pensado como una fila en una colección. Puede contener estructuras complejas como listas. Cada documento tiene un campo ID, que es usado como clave primaria y cada colección puede contener cualquier tipo de documento, pero los índices y las consultas solo se pueden hacer sobre una colección [25].

API. MongoDB tiene su propio lenguaje de consultas llamado *Mongo Query Language*. Para recuperar ciertos documentos de una colección, se crea un documento de consulta que contiene los campos con los que deben coincidir los documentos deseados. Se listan a continuación ejemplos de comandos [25]:

Inserción. `db.users.insert ({user id:”abc123”, age: 55, status:”A”})`
Selección. `db.users.find ({ status:”A”, age: 55})`
Eliminación. `db.users.remove ({ status:”A”})`

La figura 3.6 muestra la arquitectura de MongoDB.

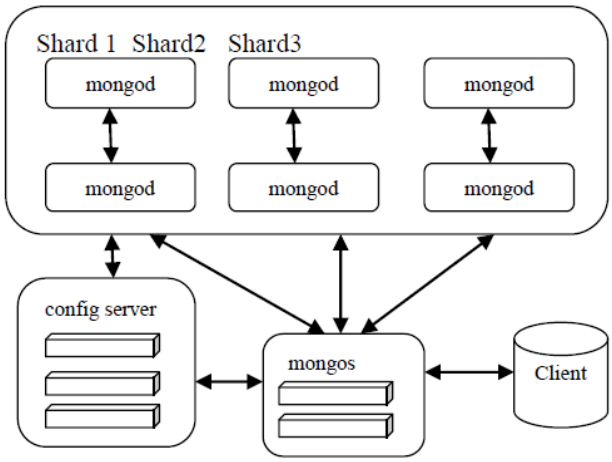

Figura 3.6. Arquitectura de MongoDB [25].

Arquitectura. El clúster de MongoDB se construye usando tres componentes principales llamados nodos shard, servidores de configuración y servicios de enrutamiento o *mongos* [25].

1. **Nodos shard:** Un clúster MongoDB utiliza uno o varios shards, donde cada uno es responsable de almacenar datos. Cada shard consiste de un nodo o de un nodo replicado que almacena datos para ese shard. Las operaciones de lectura y escritura se hacen hacia los shards apropiados.
   Un nodo replicado consiste de uno o varios servidores, donde un servidor actúa como primario y los demás actúan como servidores secundarios. Si falla un servidor primario entonces alguno de los servidores secundarios se convierte en primario. Todas las operaciones de lectura y escritura consistente van hacia el servidor primario y todas las operaciones eventualmente consistentes se distribuyen sobre los servidores secundarios [25].

2. **Servidores de configuración.** Almacenan metadatos y enrutan información en el clúster MongoDB indicando que datos se localizan en cada shard [25].

3. **Servicios de enrutamiento o mongos**. Son responsables de realizar tareas solicitadas por los clientes. Un cliente puede hacer diferentes tipos de consultas, y dependiendo del tipo de consulta, los mongos envían las solicitudes a los shard necesarios y combinan los resultados antes de ser enviados al cliente. Los mongos no tienen estado y pueden ejecutarse en paralelo [25].

3.2.2. Método propuesto

Se propone un método para la integración de operaciones del lenguaje SQL sobre una base de datos MongoDB añadiendo una interfaz entre la capa de aplicación y la capa de la base de datos. El middleware entre estas capas consiste de metadatos, los cuales comprenden diferentes paquetes.

Este sistema implementa un paquete que actúa como una interfaz entre la capa de una aplicación de JAVA y la base de datos NoSQL (MongoDB) [25]. La arquitectura de este sistema se visualiza en la figura 3.7.

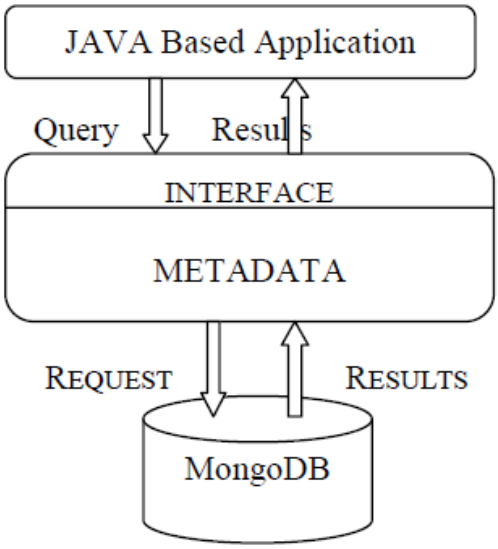

Figura 3.7. Arquitectura del sistema propuesto [25].

La aplicación de JAVA envía un comando SQL como entrada a la interfaz que actúa como middleware. La interfaz analiza la entrada y la transforma en el formato requerido por la base de datos MongoDB. La base de datos ejecuta el código reformateado y entrega resultados al middleware, el cual responde a la aplicación de JAVA [25].
El sistema hace conversión entre formatos, para lo cual almacena información para las reglas de conversión de formato y el formato de las estructuras de datos [25].
El diagrama de flujo de la figura 3.8 describe el orden de las operaciones que realiza el sistema.

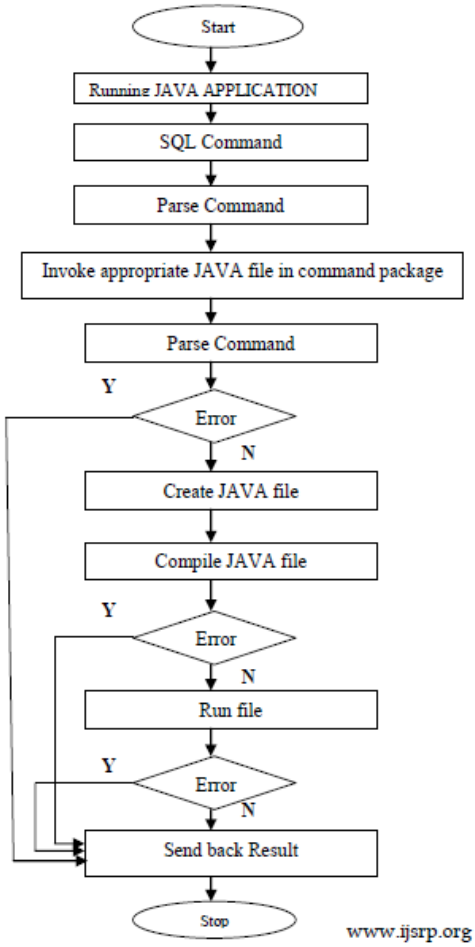

Figura 3.8. Diagrama de flujo del sistema [25].

El motivo de utilizar SQL es que se trata de un estándar muy utilizado y conocido por muchos desarrolladores. La justificación para utilizar una base de datos NoSQL como MongoDB es principalmente por cuestiones de rendimiento y manejo de un gran volumen de datos [25]. Para la aplicación es transparente tanto la complejidad de la base de datos como las reglas de conversión entre formatos especificadas en los metadatos. La aplicación solo necesita enviar consultas en SQL a la interfaz [25].

3.3. Integridad referencial y dependencias entre documentos en una base de datos orientada a documentos

Las contribuciones que se proveen en la investigación de [26] incluyen una estrategia de control de integridad referencial en una base de datos orientada a documentos bajo el enfoque de verificación. Esto es, dada una base de datos en MongoDB, se procede a la detección de errores de integridad referencial utilizando el modelo de programación MapReduce. Se asume que todos los documentos contienen los mismos campos y que los valores para dichos campos tienen tipos de datos válidos. Se asume la existencia de un mecanismo que detecte y filtre los documentos que no satisfagan esta condición [26].

3.3.1. MapReduce sobre MongoDB

Sea *A* una colección de documentos. Sea *K* un conjunto de claves y *P* un valor arbitrario. Entonces la función *Map* es cualquier función de la forma *map*: $A \rightarrow 2^{KxP}$[26].
La función *Map* es un proceso aislado enfocado en el procesamiento individual de documentos de la colección, independientemente de la información contenida en el resto de los documentos. La función *Map* es libre en cuanto a las operaciones que puede realizar con los datos de los documentos y puede producir un número de valores basados en ellos. A este proceso se le llama *emitting* y la función *Map* puede emitir cualquier número de valores, incluyendo "0" [26].
Si *d* es un documento y *1* es un campo de ese documento, entonces *d[1]* se refiere al valor del campo *1* en el documento *d*. Las salidas de todos los procesos map, $R = \cup\{map(d) \mid d \in A\}$ se agrupan bajo la clave emitida por el map para formar un número de clases [d] = {datos | (k, datos)∈R} para cada clave diferente emitida por el map [26].
Cada clase es enviada a un proceso reduce de la forma *reduce*: $K \times 2^{P} \rightarrow F$, donde F es el tipo de un resultado final. El proceso reduce es lo que "agrega" los resultados de un proceso map individual, los cuales primero son agrupados por la clave electa de la función map.
Como ejemplo, suponga que se tiene una colección de documentos de gente de ambos sexos, donde cada individuo tiene un campo que indica su sexo y un campo que indica su edad. Lo que se necesita obtener es el promedio de edad de todos los hombres y de todas las mujeres en la base de datos [26]. Se usa una función *Map* que emita la edad de cada persona y agrupe edades por sexo, y luego se usa una función *Reduce* para calcular los valores promedio de los conjuntos de valores especificados por el map [26].
Dichas funciones se pueden implementar de la siguiente forma:

```
function map () {
     emit(this.sex, this.age);
}

function reduce (key, emits) {
     return average(emits);
}
```

Donde `average (emits)` es el promedio de los números en el arreglo de edades. En MongoDB se provee de una referencia al documento mediante la variable *this* en la función *Map*, en lugar de un parámetro de función. En el ejemplo, la función *Map* produce como salida un número de tuplas `sex:age` , una para cada persona. El framework MapReduce agrupa los elementos de edad en dos clases `[male]` y `[female]`, conteniendo todas las edades de todos los hombres y de todas las mujeres, respectivamente. Finalmente se ejecuta dos veces la función *Reduce* con los argumentos `reduce (male, [male])` y `reduce (female, [female])`.
El resultado final consiste de dos tuplas `(male, am)` y `(female, af)`, donde *am* es el promedio de edad de los hombres, *y af* es el promedio de edad de las mujeres [26].

3.3.2. Verificación de relaciones uno a muchos

Sean dos colecciones de documentos *A* y *B* con sus correspondientes campos *kA* y *kB*, los cuales se consideran claves. En *A*, *kA* es clave primaria; y en *B*, *kB* es una clave que hace referencia a dicha clave. La meta de verificación de integridad referencial entre las dos colecciones es revisar si la clave *dB [kB]* para cada documento de la colección *B* existe exactamente un documento de la colección *A* con clave *dA [kA]*. En otras palabras, esto es: $\forall dB \in B \; \exists dA \in A$ tal que *dA[kA] = dB[kB]*. Esto expresa una relación uno a muchos entre registros de *A* y *B*, donde los registros en *B* refieren a registros en *A* [26].

La esencia del enfoque de verificación es usar la función *Map* para generar contadores para cada clave en *A* y *B,* agrupando todos los contadores por clave. Luego la función *Reduce* suma todos los contadores. De esta manera, los datos resultantes permiten identificar errores entre referencias de *A* y *B* [26].

La siguiente función Map se aplica a los documentos de la colección *A* [26]:

```
function mapBase(){
     emit (this["kA"],{"sums":[1,0]});
}
```

Similarmente, la siguiente función se aplica a los documentos de la colección *B* [26]:

```
function mapRef(){
     emit (this["kB"],{"sums":[0,1]});
}
```

Finalmente la función *Reduce* suma los componentes de todos los documentos que se emiten bajo la misma clave [26].

```
function reduce (key,emits){
     var sums = [0,0];
     for (var i in emits){
          sums[0] += emits[i].sums[0];
          sums[1] += emits[i].sums[1];
     }
     return {"sums":sums};
}
```

Para invocar el proceso MapReduce en MongoDB se utilizan las siguientes expresiones [26]:

```
db[A].mapReduce (mapBase, reduce, {out: {reduce: "result"}});
db[B].mapReduce (mapRef, reduce, {out: {reduce: "result"}});
```

El resultado son tuplas de la forma *tk* = (*key, countA, countB*) en la colección `result` para cada *key* única en *A* o *B*. Las tuplas contienen el número de ocurrencias de *key* como valores de *kA* y *kB* respectivamente.

La integridad referencial respecto de cada *key* puede ser verificada examinando los valores de *countA*.

- Si *countA = 0* significa que *key* existe en *B* pero no hay ocurrencias en *A*.
- Si *countA> 1* significa que *key* no es única en *A*.

Entonces, si se filtran todos los elementos en el conjunto de resultados para los cuales se cumple *countA≠ 1,* se obtiene una lista de todas las referencias incorrectas en *B*.

Ejemplo

Las colecciones A y B contienen los siguientes datos [26]:

```
A:{
{"key":"ka"},
{"key":"ka"},
{"key":"kb"}}
B:{
{"key_a":"ka"},
{"key_a":"kb"},
{"key_a":"kc"}}
```

La salida de la función *mapBase* es [26]:

```
{
{"ka":{"sums":[1,0]},
{"ka":{"sums":[1,0]},
{"kb":{"sums":[1,0]}
}
```

La salida de la función *mapRef* es [26]:

```
{
{"ka":{"sums":[0,1]},
{"kb":{"sums":[0,1]},
{"kc":{"sums":[0,1]}
}
```

El framework MapReduce hace una agrupación por clave de las salidas de *mapBase* y *mapRef*, de tal forma que los procesos reduce se ejecutan con los siguientes parámetros:

```
reduce ("ka",{
{"sums":[1,0]},
{"sums":[1,0]},
{"sums":[0,1]}})
= [2,1]
```

```
reduce ("kb", {
{"sums":[1,0]},
{"sums":[0,1]}})
= [1,1]

reduce ("kc",{ {"sums":[0,1]}}) = [0,1]
```

Estos resultados revelan problemas con *ka* y *kc* [26].

3.3.3. Verificación de relaciones muchos a muchos

En esta configuración, un documento en *B* puede referir a varios documentos en *A* y viceversa. Un uso común de este modelo es un ejemplo de libros y autores, donde cada libro puede ser escrito por múltiples autores, y cada autor puede escribir múltiples libros.
Si este ejemplo se implementara en una base de datos orientada a documentos, los documentos de una colección de autores pueden contener listas de identificadores de libro. De la misma manera, la colección de libros puede contener listas de identificadores de autores. Esta estructura introduce redundancia pero el enfoque es comúnmente usado para optimización de consultas en bases de datos [26].
Sean *dA[kA]* y *db[kB]* identificadores únicos de registro y existe un campo *refB* en cada *dA* y un campo *refA* en cada *dB*, conteniendo listas de claves en *B* y *A,* respectivamente. La meta de verificación de integridad referencial es revisar si todo miembro de todo *dA[refB]* es una clave válida en *B* y todo miembro de todo *dB[refA]* es una clave válida en A [27].
El enfoque es similar: se usa una función *Map* para contar todas las claves que refieren a documentos en *A* y *B*, pero se emiten un conjunto de valores, uno por cada elemento de la correspondiente lista de referencia [26].

La función *mapBase* se especifica como [26]:

```
function mapBase(){
     emit (this["kA"],{"sums":[1,0]});
     for (var i in this["refB"]{
          emit (this["refB"][i], {"sums":[0,1]});
     }
}
```

De forma similar se especifica la función *mapRef* [26]:

```
function mapRef(){
     emit (this["kB"],{"sums":[1,0]});
     for (var i in this["refA"]{
          emit (this["refA"][i], {"sums":[0,1]});
     }
}
```

El primer miembro que se emite es el contador de la clave primaria, mientras que el segundo miembro cuenta las claves usadas como referencias.
La función reduce no cambia [26]:

```
function reduce (key,emits){
     var sums = [0,0];
     for (var i in emits){
          sums[0] += emits[i].sums[0];
          sums[1] += emits[i].sums[1];
     }
     return {"sums":sums};
}
```

El resultado del proceso son tuplas de la forma *tkey = (key, countkey, countref)* y examinando los valores de las tuplas se revelan las siguientes inconsistencias [26]:

- Si *countkey = 0* significa que *key* está presente en la referencia de algún registro, pero no existe una clave primaria en la colección referenciada.
- Si *countkey > 1* significa que *key* no es una clave única en la colección correspondiente.

3.3.4. Conclusiones

Cuando la capa de base de datos de un sistema no provee mecanismos para asegurar la integridad referencial de los datos, problemas con el código del programa y transacciones incompletas pueden causar serios errores para el sistema. En teoría, cuando la correctez del programa es completamente probada, los casos de inconsistencia se reducen al mínimo. Sin embargo, todo sistema se puede beneficiar de un mecanismo de verificación de sus datos para manejar problemas no detectados con la lógica del programa o fallas de otros tipos, las cuales causan que operaciones de actualización creen inconsistencias en los datos [26].
El enfoque propuesto permite detectar fallas relativas a referencias hacia claves primarias no existentes y claves primarias duplicadas.
El mecanismo propuesto sólo reporta la existencia de esos problemas, no hace ninguna sugerencia acerca de las causas de la generación de documentos inconsistentes. Sin embargo, si la verificación se ejecuta periódicamente, puede ser un instrumento importante para la detección temprana de problemas con el código y su pronta solución, antes de que los problemas se repliquen al resto del sistema [26].

3.4. Resumen

Este capítulo presenta investigaciones relacionadas a la propuesta de esta tesis. Es de importancia enfatizar que dos de estos trabajos se enfocan en el control de integridad referencial de una base de datos NoSQL, mientras que en esta tesis el enfoque es hacia el control de la integridad de dominio. El capítulo 4 explica y detalla las propuestas de esta tesis.

En los tres trabajos relacionados, se utiliza algún tipo de DBMS no relacional. En [24] se trata de Cassandra, que almacena columnas; mientras que en [25] y [26] se trata de MongoDB, el cual almacena documentos. Las investigaciones de [24] y [25] engloban la idea de una capa intermedia de software entre una base de datos y un programa de aplicación, la cual enriquece la funcionalidad de la base de datos. Esta capa intermedia es un API experimental en [24] y *metadatos* en [25]. La diferencia es que en [25] no se controla ningún tipo de integridad, sino que es un mapeo de SQL a sentencias de MongoDB. Otra diferencia importante y sutil es que el significado de *metadatos* es muy diferente en estos trabajos. Para esta tesis, el concepto de *metadatos* es el mismo que en [24]. Esto es, datos que especifican restricciones de integridad.

El control de integridad referencial de [24] es desde un enfoque de prevención sobre una base de datos orientada a columnas, mientras que la estrategia de [26] es de verificación sobre una base de datos documental.

Capítulo 4. Implementación de restricciones de integridad en una base de datos NoSQL documental

Este capítulo describe en detalle, las estrategias empleadas para la implementación de restricciones de integridad de dominio en una base de datos NoSQL documental. Esto cubre el modelo de representación de metadatos, los algoritmos que posibilitan la validación de la información y las características de optimización. Dichas estrategias son las contribuciones de esta tesis. La obtención del software que implementa los algoritmos descritos permite extender la funcionalidad de la base de datos documental, y de esta manera, ésta puede adecuarse a aplicaciones que requieren de integridad, aún aprovechando los beneficios de NoSQL.
El desarrollo de este proyecto utiliza el DBMS MongoDB y el lenguaje de programación JAVA.

4.1. Descripción general de las estrategias

Las estrategias empleadas para la implementación de restricciones de integridad en una base de datos documental consisten de:

1. La especificación y almacenamiento de *metadatos*.
2. Un programa intermedio entre la aplicación de usuario y la base de datos (*interfaz de integridad*).

Los *metadatos* son documentos especiales en la base de datos que describen cómo deben ser los documentos de datos en términos de restricciones de integridad asociados a ellos.
El programa intermedio, llamado *interfaz de integridad,* hace uso de estos metadatos cada vez que la aplicación de usuario necesita escribir datos, de tal manera que se asegure la integridad de los mismos. Este programa intermedio juega un rol parecido al API que se propone en [24].
Durante una inserción o actualización, la interfaz de integridad primero debe hacer una consulta a la base de datos para obtener los metadatos que se necesitan. Una vez obtenidos, se hace un proceso de validación de los datos que se quieren insertar o actualizar, en base a las restricciones definidas en los metadatos. Si un documento es incorrecto, entonces la interfaz de integridad lo rechaza e informa a la aplicación de usuario acerca de los campos donde existen problemas. Cuando el documento es correcto, éste es insertado o actualizado en la base de datos, y se informa a la aplicación de usuario con un mensaje de éxito.
La colección donde se insertan o actualizan datos puede ser muy grande, pero es posible obtener rápidamente los metadatos sin necesidad de hacer una consulta que revise toda la colección. Esto es gracias a la indexación, la cual permite recuperar los metadatos de una estructura de datos especial que favorece la eficiencia.

4.2. Representación de metadatos

4.2.1. Metadatos de un conjunto de restricciones

Los metadatos son representados mediante documentos especiales que obligatoriamente tienen el campo *metadata*, el cual sirve para indicar el campo al cual se aplican ciertas restricciones.

Estas restricciones son las siguientes:

- **Tipo de dato:** Especifica cuál es el tipo de dato esperado que se va a almacenar. Puede tratarse de enteros, números reales, fechas, cadenas o caracteres simples.
- **Rango**: Especifica un intervalo de valores para los cuáles es válido el dato. Esto solo aplica para números y fechas.
- **Longitud**: Especifica el mínimo y el máximo número de dígitos que el valor puede tener, en el caso de números. En el caso de cadenas determina el mínimo y el máximo número de caracteres.

Un documento de metadatos para un campo está compuesto por el nombre del campo y los tres tipos de restricciones descritas. Visualmente esto equivale a lo siguiente:

Metadata: Nombre del campo	Tipo de dato	Rango	Longitud

Figura 4.1. Estructura de un documento de metadatos para un conjunto de restricciones.

La estructura sugerida es la misma para todos los campos donde está definida alguna restricción. Si ninguna restricción aplica a un campo, entonces no es necesario hacer ninguna validación y se procede a revisar los demás campos. En dicho caso no existe un documento de metadatos para dicho campo. También es posible que un campo tenga definida alguna o algunas restricciones, no necesariamente todas. En esa situación las restricciones que no apliquen no se guardan en el documento de metadatos. De hecho lo común es usar la restricción de rango o la de longitud, pero no ambas, porque en el caso de enteros una se infiere de la otra. En el caso de cadenas solo se utiliza longitud y las fechas solo ocupan el rango. Entonces el único tipo de dato que puede beneficiarse de ambas restricciones son los números reales, donde puede valer la pena especificar cierto grado de precisión de cifras decimales y para un rango dado. Como las colecciones carecen de esquema y los documentos pueden tener diferentes campos, es posible guardar un documento de metadatos que contenga sólo las restricciones que apliquen al campo al que se referencia.

Un documento en una colección se considera íntegro solo si todos los valores de sus respectivos campos son válidos respecto del conjunto de restricciones que apliquen a cada uno de ellos. Una colección de documentos está íntegra solo si todos los documentos que contiene son íntegros. Finalmente, una base de datos documental se encuentra íntegra solo si todas las colecciones en las que existen restricciones están íntegras. Esta definición considera la posibilidad de que para algunas colecciones sea inexistente alguna restricción de integridad respecto de cualquier campo.

Ejemplo

Suponga que se tiene una base de datos que almacena información de computadoras. En una colección se almacenan datos del fabricante del equipo, así como la frecuencia del microprocesador, el total disponible de memoria RAM, el número de serie y la fecha de adquisición del equipo.

Un documento de esta colección tiene la siguiente estructura:

Fabricante	Frecuencia del micro	RAM	Número de serie	Fecha de adquisición

Considere las siguientes restricciones:

- Los equipos que se venden tienen un máximo de RAM de 9 GB y además las frecuencias de microprocesador siempre oscilan entre 1.8 Ghz y 3 Ghz.
- El nombre del fabricante es una cadena entre 4 y 10 caracteres.
- El número de serie es una cadena de diez caracteres.
- Las fechas de adquisición siempre son posteriores al primero de enero de 2014.

Con estas restricciones es posible especificar los siguientes documentos de metadatos para la colección dada:

```
{metadata: "fabricante", tipo: "cadena", longitud: {min: 4, max:10}}
{metadata: "frecuencia", tipo: "real", rango: {min:1.8, max:3}}
{metadata: "RAM", tipo: "entero", longitud: {max:1}}
{metadata: "num_serie", tipo: "cadena", longitud: {max:10}}
{metadata: "fecha_adquisicion", tipo: "fecha", rango: {min: 1/1/2014}}
```

El ejemplo ilustra que pueden estar definidas una o varias restricciones para cada campo almacenado. Cuando una restricción está en desuso, entonces no existe ese campo en el documento. Es el caso del campo *frecuencia* que no tiene *longitud*.
Los documentos de metadatos sugeridos siguen la sintaxis de MongoDB y aprovechan su modelo de datos para guardar subdocumentos en el caso de los campos *longitud* y *rango*.

4.2.2. Metadatos de restricción por conjunto de valores

Es posible requerir una restricción más estricta respecto de los datos que se pueden almacenar que el tipo de dato, la longitud y el rango. Si éste es el caso, entonces se utiliza otro tipo de restricción que especifica un conjunto de posibles valores que el dato puede tomar. Cualquier valor que no coincida con alguno de los valores del conjunto es rechazado por la base de datos. La figura 4.2 muestra la estructura de este tipo de metadato.

Metadata:Nombre del Campo	Lista de valores:[]

Figura 4.2. Estructura de un documento de metadatos para una restricción por conjunto de valores.

El campo *lista de valores* puede guardarse como un arreglo o lista que es consultado cada vez que se inserta o actualiza un dato.

Ejemplo

Suponga que ahora en la colección que almacena información de computadoras solo pueden existir tres fabricantes específicos: ACER, DELL y HP.
Esto da lugar al siguiente documento de metadatos:

```
{metadata: "fabricante", lista de valores: ["ACER", "DELL", "HP"]}
```

Al insertar o actualizar en la base de datos el nombre del fabricante, éste debe coincidir con algún valor previamente definido en el campo *lista de valores* de este documento de metadatos.

4.2.3. Metadatos para la restricción de campos obligatorios

Aunque las colecciones de documentos carecen de estructura, en algunos casos es necesario que los documentos incluyan ciertos atributos de los cuales depende el funcionamiento de una aplicación o de todo un sistema. Es en esos escenarios donde se necesita introducir un tercer tipo de metadato, el cual establece que todos los documentos de una colección deben incluir los campos o atributos previamente definidos, y dichos atributos deben tener un valor. A diferencia de los anteriores tipos de metadatos, la restricción de campos obligatorios se define en el nivel de colección y solo existe un documento de metadatos de este tipo por colección (siempre que aplique). La estructura de este tipo de metadatos es la siguiente:

`Metadata: "requerido"`	`Requerido: []`

Figura 4.3. Estructura de un documento de metadatos para la restricción de campos obligatorios.

El valor *requerido* en el campo *Metadata* indica que se trata de un documento que sirve para restringir campos obligatorios. El campo *Requerido* es un arreglo o lista que contiene los nombres de todos los atributos o campos que un documento de datos debe contener, y que además esos atributos deben tener algún valor.

Ejemplo

Considere el ejemplo anterior de los datos de equipos de cómputo. Suponga que el número de serie es un campo obligatorio de todos los documentos, pues éste sirve como un identificador. Entonces puede establecerse el siguiente documento de metadatos:

```
{metadata: "requerido", requerido: ["num_serie"]}
```

En este caso el arreglo *requerido* solo contiene un elemento, pero en la práctica puede tener cualquier cantidad de nombres de atributos obligatorios.

4.3. Indexación de metadatos

4.3.1. Justificación e implicaciones de la indexación

Para la interfaz de integridad, es imprescindible el rápido acceso a los metadatos, considerando que cada inserción y actualización en una colección donde existen restricciones los requiere para hacer validaciones. Sería impráctico e ineficiente revisar toda una colección de documentos para encontrar los metadatos, pues ésta puede ser inmensa y el promedio de escrituras puede ser grande.
La indexación pone solución al problema de la consulta eficiente de metadatos, y su efecto es el mismo que en otros tipos de bases de datos: la reducción de costosas lecturas al disco duro. Entonces, cuando se hacen consultas para obtener metadatos, el sistema de MongoDB tan sólo revisa el índice y lo utiliza para recuperar dichos documentos. Las implicaciones de usar índices en MongoDB es que consumen espacio en disco y memoria, cada uno con un tamaño de 8 KB. Además es necesario actualizarlos por cada inserción o actualización, de modo que su uso es preferible en escenarios donde el promedio de lecturas supera al de escrituras [30]. Otra implicación de los índices en MongoDB es que su construcción bloquea todas las operaciones de la base de datos. Actualmente ya existe un remedio para esto mediante una opción en el comando de creación del índice. Cuando se habilita esta opción, la creación del índice se ejecuta en el background y la instancia de MongoDB puede continuar con operaciones de lectura y escritura. Esta creación en el background utiliza una aproximación incremental y es más lenta. Además, las consultas no utilizan índices parcialmente construidos, sólo están disponibles hasta que se completen.

4.3.2. Descripción de los índices de metadatos

Durante el proceso de validación de documentos, la interfaz de integridad hace consultas de los metadatos en base al campo *metadata*. Dicho campo guarda el nombre del atributo para el cual están definidas ciertas restricciones, las cuales pueden incluir *tipo*, *rango*, *longitud* o *lista de valores*, según se necesite. Entonces una consulta de los metadatos incluye el campo *metadata* en los criterios de búsqueda y una proyección de todos los demás campos en los documentos de metadatos.
En el ejemplo de los datos de computadoras, esto equivale a lo siguiente cuando se desea obtener los metadatos del campo frecuencia:

```
db.computadoras.find({metadata:"frecuencia"}, {metadata:0})
```

Donde lo primero que se encuentra entre llaves son criterios de búsqueda y después de la coma aparece la proyección de la consulta. Por simplicidad, es mejor indicarle al DBMS que excluya de la proyección *metadata* que listar todos los campos del documento de metadatos. Además, como no todos los documentos de metadatos tienen necesariamente los mismos campos, no es posible listar en la proyección todos los posibles campos.

Para que un índice cubra todas las consultas de los metadatos, es necesario que éste contenga tanto los atributos de búsqueda como los de proyección. En el ejemplo, esto da lugar a la creación del siguiente índice compuesto, suponiendo que se necesitan metadatos por conjunto de restricciones:

```
db.computadoras.ensureIndex({metadata:1, tipo:1, rango:1, longitud:1})
```

Si los documentos se ordenan ascendente o descendentemente es irrelevante, porque lo único que interesa es mantener una estructura especial y más pequeña a la colección de donde se consultan los metadatos. Tampoco es importante si en este índice se listan en otro orden los campos de restricción. Sin embargo, lo que es imperativo es que el campo *metadata* siempre se liste en primer lugar, porque es un campo que tienen obligatoriamente todos los documentos de metadatos, a diferencia de los otros tres.
Otro índice se usa en caso de que se necesiten metadatos de restricciones por conjunto de valores. Una consulta a la base de datos de este tipo de metadatos tiene exactamente la misma estructura que si se tratara de metadatos de un conjunto de restricciones. Lo que es diferente son los campos del índice, que en este caso contiene el campo *metadata* y el campo *lista de valores*.
Volviendo al ejemplo de los datos de computadoras y tomando en cuenta que el campo *fabricante* admite sólo tres posibles valores, esto da lugar al siguiente índice:

```
db.computadoras.ensureIndex({metadata:1, lista de valores:1})
```

En este tipo de índice también es necesario que el campo *metadata* se liste en primer lugar, porque las consultas se hacen en base a este campo y no en base al campo *lista de valores*, el cual se agrega como parte del índice solo porque forma parte de la proyección en una consulta de metadatos. Nuevamente es irrelevante el ordenamiento interno de los documentos.

4.4. Implementación de restricciones de integridad de dominio

4.4.1. La interfaz de integridad

La interfaz de integridad es el componente de software que se encarga de la correcta inserción y actualización de los documentos en la base de datos. Mediante esta capa intermedia, la aplicación de usuario suministra los documentos de metadatos necesarios para validar integridad. Idealmente esto es lo primero que se hace. Sin embargo, la interfaz de integridad no se utiliza para verificar documentos que ya están en la base de datos, porque es un enfoque preventivo. La figura 4.4 muestra el diagrama de bloques que representa la interacción entre la aplicación de usuario, la interfaz de integridad y la base de datos.

Figura 4.4. Diagrama de bloques de la interfaz de integridad y su entorno.

Una vez que la base de datos contiene metadatos, la interfaz de integridad los consulta cada vez que se van a insertar los documentos que proporciona la aplicación, de tal forma que se

valide su integridad. Luego de hacer la validación, la interfaz informa a la aplicación acerca del éxito de la operación de inserción o actualización. Solo si los documentos son correctos respecto de las restricciones asociadas a sus campos, éstos se insertan en la base de datos.
Los criterios de actualización de documentos se verifican de la misma manera y solo se aplican si son correctos.
La interfaz de integridad y la aplicación de usuario residen en el cliente. La interfaz de integridad hace la conexión a una base de datos remota que está en el servidor y envía todas las operaciones de lectura y escritura. Los documentos a insertar se envían formateados con la sintaxis de JSON hacia el servidor.
También se da soporte a las operaciones de inserción y actualización de documentos sin verificación, además de consultas y eliminaciones en la base de datos. La indexación de los metadatos también es una función de la interfaz, creando los índices necesarios de forma dinámica mientras la aplicación suministra los metadatos.

4.4.2. Algoritmo principal de la interfaz de integridad

Todos los documentos de datos que la interfaz de integridad recibe como entrada, son procesados de la misma manera. La función del algoritmo principal es la verificación de la integridad de un documento. La aplicación de usuario utiliza este algoritmo las veces que necesite, permitiendo la validación de múltiples documentos.
En el apartado 4.4.3 se describe formalmente este algoritmo en pseudocódigo, llamado *insertDoc*. Por ahora, la prioridad es tener una visión general de este algoritmo, el cual está compuesto por llamadas a métodos específicos de validación, dentro de la interfaz de integridad. La figura 4.5 muestra visualmente un diagrama de flujo que ilustra la operación general de *insertDoc*.

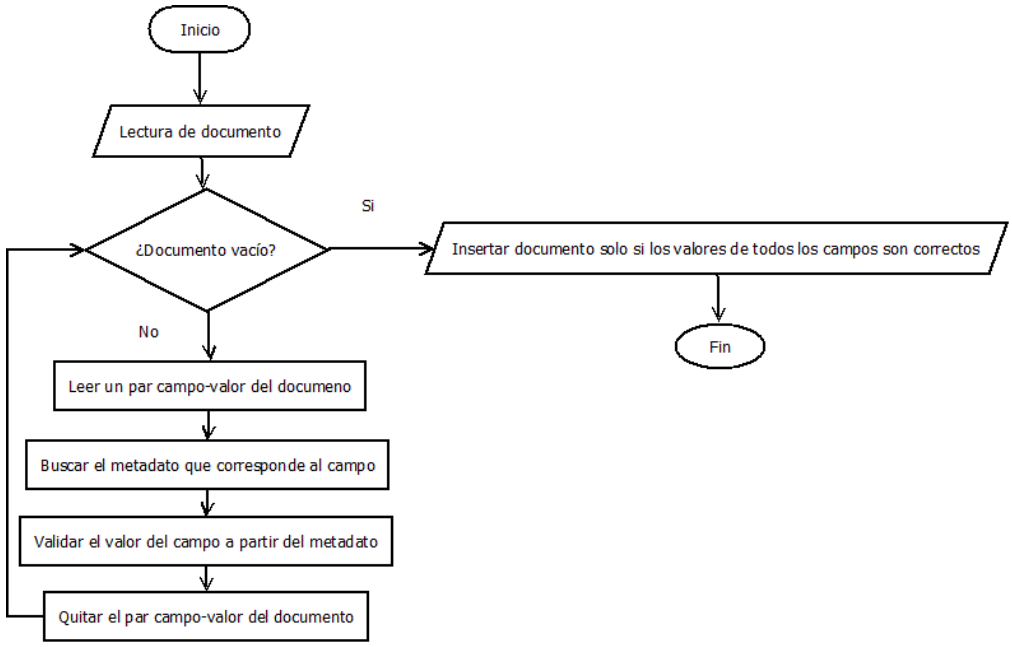

Figura 4.5. Diagrama de flujo del algoritmo principal de la interfaz de integridad.

El diagrama de flujo resume la operación del algoritmo principal, y su lógica es la misma que para validar documentos de actualización. Por simplicidad, el diagrama quita elementos del documento. Sin embargo, en la implementación, el proceso iterativo no quita pares campo-valor del documento.

4.4.3. Funciones de la interfaz de integridad

En los siguientes apartados se muestran los algoritmos en pseudocódigo que componen la interfaz de integridad. Por simplicidad, claridad y practicidad se omiten muchos detalles irrelevantes de implementación y se abrevia sintaxis, con el objetivo de ilustrar solo lo más importante de dichos algoritmos. La interfaz comprende los siguientes métodos, los cuales hacen operaciones específicas en la base de datos.

- `void insertMetadata(collection, field, list)`: Inserta metadatos para un campo al cual se aplican restricciones por conjunto de valores, dados por la lista del parámetro. Crea dinámicamente un índice para este tipo de restricción.

- `void insertMetadata(collection, field, type, range, length)`: Inserta metadatos para un campo al cual se aplican un conjunto de restricciones. Los parámetros tipo, rango y longitud pueden ser nulos y su uso depende de lo que se requiera para ese campo. Crea dinámicamente un índice para este tipo de restricción.

- `void insert(collection, document)`: Inserta un documento en la base de datos sin ser verificado por la interfaz.

- `void update(criteria, action)`: Actualiza un conjunto de documentos en la base de datos con la acción especificada para los documentos que cumplan el criterio de actualización. No se verifica si la actualización es válida.

- `String insertDoc(collection, document)`: Inserta un documento en la base de datos solo si sus campos son correctos respecto de las restricciones asociadas a ellos. Lee los metadatos necesarios de la base de datos para hacer la validación. Regresa un mensaje de confirmación.

- `String updateDoc(collection, criteria, action)`: Actualiza los documentos que cumplan con el criterio de actualización solo si la acción de actualización es válida respecto de las restricciones que aplican a los campos de los documentos asociados. Lee los metadatos necesarios de la base de datos para hacer la validación. Regresa un mensaje de confirmación.

- `int remove(collection, criteria)`: Elimina de la colección especificada los documentos que cumplan con el criterio de eliminación. Regresa la cantidad de documentos eliminados.

- `boolean validate (metadata, value):` Dado un documento de metadatos y el valor asociado al campo correspondiente, verifica que el valor sea válido y regresa verdadero en caso afirmativo, falso en caso contrario.
- `boolean validateSET(metadata, value):` Su función es la misma que `validate`, pero solo aplica a campos restringidos por un conjunto de valores.

4.4.4. Inserción de metadatos

Las funciones de la interfaz relacionadas con la inserción de metadatos son:

```
1. void insertMetadata (collection, field, list)
2. void insertMetadata(collection, field, type, range, length)
```

A continuación se describe el algoritmo para la primera de estas funciones:

```
void insertMetadata (collection, field, list)
      metadata = new document("metadata": field)
      metadata.add("set",list)
      db.collection.insert(metadata)
      if(db.metadataIndex not exists) then
            db.createIndex(metadata)
```

La función crea un documento de metadatos con el campo "metadata" y el nombre del campo como valor. Luego agrega un segundo par campo-valor llamado *set*, una lista que contiene el conjunto de posibles valores que puede tomar ese campo. El documento se inserta en la colección especificada en el parámetro.
A continuación se describe el algoritmo para la segunda de estas funciones:

```
void insertMetadata(collection, field, type, range, length)
      metadata = new document("metadata": field)
      if(type not null) then
            metadata.add("type", type)
      if(range not null) then
            minRange = range.min
            maxRange = range.max
            metadata.add("range", minRange, maxRange)
      if(length not null) then
            minLength = length.min
            maxLength = length.max
            metadata.add("length", minLengt, maxLength)
      db.collection.insert(metadata)
      if(db.metadataIndex not exists) then
            db.createIndex(metadata)
```

La función inserta en la colección especificada en el parámetro el nuevo documento de metadatos con los campos que apliquen. Cada campo es un tipo de restricción que aplica al campo especificado en el parámetro. Por simplicidad y para abreviar código se insertan longitudes y rangos mínimos y máximos, pero alguno de esos valores puede ser nulo. En la implementación real esto es verificado antes de agregar ese campo en el documento de metadatos. La base de datos guarda como un documento embebido los máximos y mínimos,

como se muestra en los ejemplos de la sección 4.2. Ambas funciones crean un nuevo índice en la base de datos sobre el campo *metadata*, si es que no existía.

4.4.5. Inserción de documentos

Las funciones de la interfaz relacionadas con la inserción verificada de documentos son las siguientes.

1. `boolean validate(metadata, value)`
2. `boolean validateSET(metadata, value)`
3. `String insertDoc(collection, document)`

Las primeras dos funciones listadas dan soporte a las operaciones de inserción (*insertDoc*) y actualización (*updateDoc*) verificadas. Su tarea es revisar que el valor del campo de un documento es correcto respecto de las restricciones previamente especificadas en el documento de metadatos correspondiente. Cuando terminan su procesamiento regresan un valor lógico indicando si el valor es válido o no lo es. La función *insertDoc* hace una llamada a la función correspondiente por cada campo del documento. Su función es aplicar la inserción si es correcta e informar a la aplicación sobre el éxito o fracaso de la operación.
A continuación se describe el algoritmo de la función *validate*.

```
boolean validate (metadata, value)
	if(metadata.containsField(“type”) then
		type = metadata.get(“type”)
	if(metadata.containsField(“range”) then
		range = metadata.get(“range”)
		minRange = range.min
		maxRange = range.max
	if(metadata.containsField("length") then
		length = metadata.get(“length”)
		minLen = length.min
		maxLen = length.max
	if(type = “int”) then
		if(value NOT INTEGER) then
			return false
	else if(type = “real”) then
		if(value NOT REAL) then
			return false
	else if(type = “char”) then
		if(value.size > 1) then
			return false
	else if(type = “date”) then
		day = value.day
		mon = value.month
		year = value.year
		if(day < 1 OR day >31 OR mon < 0 OR mon > 12 OR year < 0) then
			return false
	if(value < minRange OR value > maxRange) then
		return false
	if(value < minLen OR value > maxLen) then
		return false
	return true
```

El algoritmo engloba la idea de validar un campo usando todas las restricciones de dominio que apliquen, pues dependiendo del tipo de dato y del uso, puede no existir o ni siquiera tener sentido algún tipo de restricción. Ante cualquier error en el valor del campo respecto de las longitudes, rangos y tipo de dato, la función regresa falso. Si al final el valor del campo pasa todas las pruebas entonces es correcto y la función regresa verdadero.
La descripción de este algoritmo ha omitido muchos detalles para facilitar su entendimiento. En la implementación real se verifica la existencia de mínimos y máximos, las fechas tienen un procesamiento más largo y los rangos y longitudes se validan de una manera específica para cada tipo de dato.
A continuación se describe el algoritmo de la función *validateSET*.

```
boolean validateSET(metadata, value)
	list = metadata.get(“set”)
	for(Element e: list)
		if(value = e)
			return true
	return false
```

En el tipo de restricción por conjunto de valores, se revisa que el valor del elemento a insertar coincida con alguno de los valores previamente definidos en la lista del documento de metadatos. Si existe una coincidencia la función regresa verdadero. Si no existen coincidencias entonces el valor no es correcto y la función regresa falso.

A continuación se describe el algoritmo de la función *insertDoc.*

```
String insertDoc(collection, document)
	correct = true
	correctAll = true
	for(Element e: document)
		field = e.getField()
		value = e.getValue()
		metadata = db.query(“metadata”: field)
		if(metadata not null) then
			if(metadata.containsField(“set”) then
				correct = validateSET(metadata, value)
			else
				correct = validate(metadata, value)
		correctAll = correctAll AND correct
	if(correctAll = true) then
		db.collection.insert(document)
		return “document inserted”
	else
		return “ERROR in document”
```

La entrada a esta función es el nombre de la colección en la cual se inserta el documento, el cual debe verse como un conjunto de pares campo-valor. Un *elemento* es un par campo-valor que existe dentro del documento. La función hace una consulta a la base de datos para obtener el documento de metadatos que se necesita para validar el *elemento*. Esto lo hace en base al campo del *elemento*, el cual en la consulta es en realidad el valor del campo “*metadata*”.

Cuando no existe un documento de metadatos para el elemento actual, entonces no es necesario hacer ninguna validación.
Si el documento de metadatos contiene el campo "set", significa que la validación es por conjunto de valores, de modo que se invoca al método adecuado. Por simplicidad, el mensaje de error no muestra detalles, pero en la implementación la interfaz informa claramente qué campos tienen errores. Sólo cuando todos los elementos del documento son correctos, el nuevo documento se inserta en la base de datos y la función regresa un mensaje de éxito.

4.4.6. Actualización de documentos

Las funciones de la interfaz relacionadas con la actualización verificada de documentos son las siguientes.

1. `boolean validate (metadata, value)`
2. `boolean validateSET(metadata, value)`
3. `String updateDoc(collection, criteria, action)`

Las funciones *validate* y *validateSET* funcionan de la misma manera que para verificar la inserción correcta de documentos.

A continuación se describe el algoritmo de la función *updateDoc*.

```
String updateDoc(collection, criteria, action)
       correct = true
       correctAll = true
       for(Element e: action)
              field = e.getField()
              value = e.getValue()
              metadata = db.query("metadata", field)
              if(metadata not null) then
                     if(metadata.containsField("set") then
                            correct = validateSET(metadata, value)
                     else
                            correct = validate(metadata, value)
              correctAll = correctAll AND correct
       if(correctAll = true) then
              db.collection.update(criteria, action)
              return "documents updated"
       else
              return "Update is invalid"
```

El algoritmo de actualización se basa en lo mismo que el algoritmo de inserción. Lo único diferente son los parámetros y la operación a aplicar. Es importante señalar que tanto los *criterios* como las *acciones* de actualización son documentos que debe proporcionar la aplicación, los cuales siguen las mismas convenciones que un documento de datos típico, con diferencias sutiles. La interfaz asume la correcta sintaxis de estos documentos, y lo que verifica es su contenido semántico desde el punto de vista de las restricciones de integridad. Como la *acción* es un documento, es posible iterarla y hacer las mismas operaciones que se hacen en cualquier documento. Esta acción de actualización debe ser tal que no comprometa la integridad de los documentos, por lo que se verifica de igual manera que una inserción.

Cuando la acción es válida, se aplica en base a los criterios definidos, los cuales definen una consulta que especifica un subconjunto de la colección. Al terminar la verificación, la función regresa un mensaje que confirma el éxito o el fracaso de la actualización.

4.4.7. Inserción y actualización sin verificación.

La interfaz de integridad da soporte a las operaciones de inserción y actualización de documentos sin verificación. Esto es, los documentos que suministra la aplicación se insertan directamente en la base de datos, sin hacer ningún tipo de validación. Lo mismo puede decirse de la función de actualización sin verificación. Las funciones de la interfaz relacionadas con la inserción y actualización sin verificación son las siguientes:

```
1. void insert(collection, document)
2. void update(criteria, action)
```

El algoritmo de la función de inserción es el siguiente:

```
void insert(collection, document)
      db.collection.insert(document)
```

El algoritmo de la función de actualización es el siguiente:

```
void update(criteria, action)
      db.collection.update(criteria, action)
```

Ambas funciones aplican directamente la operación correspondiente usando los parámetros que proporciona la aplicación.

4.4.8. Eliminación de documentos.

La función de eliminación de documentos borra de la colección especificada aquellos documentos que cumplan con los criterios de eliminación. Regresa el número de documentos eliminados.

```
int remove(collection, criteria)
      db.collection.remove(criteria)
      return num_docs_removed
```

4.5. Optimizaciones de la interfaz de integridad

4.5.1. Necesidad de la optimización

Con el principal objetivo de que la interfaz de integridad sea lo más eficiente posible, se describen a continuación dos características para su optimización.
La aplicación de usuario suministra una cantidad de documentos que necesitan ser actualizados o insertados, todos ellos independientes entre sí. Esto es, la verificación de los campos de un documento por la interfaz de integridad es una tarea completamente susceptible de traslaparse en tiempo de ejecución con la verificación de otros documentos.

Una solución mediante procesamiento paralelo permite la ejecución de múltiples hilos, donde cada uno se encarga de verificar cierta cantidad de documentos, aplicar la operación correspondiente (inserción o actualización) si es válida respecto de las restricciones definidas, e informar a la aplicación de usuario sobre los resultados. Como una gran parte del código de la interfaz de integridad puede ejecutarse en paralelo, vale la pena invertir tiempo y esfuerzo en buscar una solución óptima por este medio. Además, el paralelismo es importante porque el volumen de documentos a insertar puede ser muy grande, y en esos casos el procesamiento secuencial es insuficiente.

En la figura 4.4 se muestra que la interfaz de integridad necesita leer metadatos cada vez que ocurre una operación de inserción o actualización. Esto implica que primero es necesario hacer una consulta a la base de datos remota para obtener los documentos de metadatos. Si la interfaz de integridad realmente se implementa de esta manera, esto sería un cuello de botella en todo el sistema. El motivo es que se ejecuta una consulta a la base de datos por cada campo de cada documento donde existan restricciones. La suma total de estas consultas puede crecer a miles o millones, introduciendo demasiados retardos por la transmisión mediante la red y lecturas al disco duro del servidor remoto. Eso retrasa severamente la operación de la interfaz y del DBMS. Pero esta situación es aún peor: los documentos de metadatos para la misma colección y los mismos campos son exactamente los mismos y se están leyendo una y otra vez. Aun cuando los metadatos están indexados y el programa es paralelo, la operación de lectura de metadatos sigue siendo muy costosa y debe limitarse. Para solucionar este problema se propone mantener un caché de metadatos en la memoria principal de la máquina cliente que ejecuta la aplicación de usuario y la interfaz de integridad. Puede asignarse un tamaño máximo del caché, de tal forma que tampoco se comprometa la memoria del cliente de la base de datos. Este parámetro lo da la aplicación.

4.5.2. Paralelismo de la interfaz de integridad

En general, la independencia total entre dos o más procesos representa una oportunidad de paralelizar dichos procesos. En el presente contexto, la validación de un documento nunca toma como entrada datos de la validación de otro documento, motivo por el cual son tareas susceptibles de ejecutarse concurrentemente sin alterar los resultados esperados.
Esto es, la salida de la verificación de dos o más documentos es la misma si se ejecutan de manera secuencial que ejecutarlos de forma concurrente. Además, en la base de datos no interesa el orden en que se insertan documentos porque eso no aporta información.
La verificación de una actualización también puede ejecutarse concurrentemente con la de otras actualizaciones, incluso inserciones, porque el DBMS se encarga de hacer el bloqueo necesario para mantener la base de datos consistente. En MongoDB, el bloqueo es a nivel de documento. Entonces, si una actualización afecta a un documento en un instante dado, cualquier otra actualización que intente modificarlo se bloquea y espera a que termine la anterior.
Cuando la interfaz de integridad recibe un volumen de datos de entrada, primero reparte equitativamente los documentos entre un conjunto de hilos de procesamiento y éstos aplican las funciones `insertDoc()` y `updateDoc()` a los documentos que les corresponden.

4.5.3. El caché de metadatos

La segunda característica de optimización de la interfaz requiere almacenar en una estructura especial, los metadatos recuperados de la base de datos. Dicha estructura de datos se comparte entre todos los hilos concurrentes que procesan documentos, y el lenguaje de programación JAVA facilita la creación y manejo de una estructura eficiente.
En la implementación del caché, se ha seleccionado la tabla hash como la estructura de almacenamiento. Los motivos detrás de esto es que las inserciones y eliminaciones son de orden O (1) en el peor de los casos. Las búsquedas son de orden O (1) en el caso promedio y O(n) en el peor de los casos [31].
La tabla hash está compuesta por slots, donde la clave de cada slot es igual al nombre del campo asociado al documento de metadatos almacenado. Estos metadatos son recuperados de la base de datos y siguen las convenciones descritas en la Sección 4.2.

La figura 4.6 muestra una vista de la tabla hash que contiene los documentos de metadatos para el ejemplo de la Sección 4.2.

Clave	*Valor*
fabricante	{metadata: "fabricante", tipo: "cadena", longitud: {min: 4, max: 10}, timestamp:1}
frecuencia	{metadata: "frecuencia", tipo: "real", rango: {min: 1.8, max: 3}, timestamp: 2}
RAM	{metadata: "RAM", tipo: "entero", longitud: {max: 1}, timestamp:5}
num_serie	{metadata: "num_serie", tipo: "cadena", longitud: {max: 10}, timestamp: 3}
fecha adquisición	{metadata: "fecha adquisición", tipo: "fecha", rango: {min: 1/1/2014}, timestamp: 4}

Figura 4.6. Ejemplo del contenido del caché implementado como tabla hash.

Cuando la interfaz de integridad necesita acceder a los metadatos de un campo, primero revisa si ya existen en la tabla hash, usando como clave de búsqueda el nombre de ese campo. Si no se encontró el documento de metadatos, entonces es necesario hacer una consulta a la base de datos.
Cada vez que se recupera un documento de metadatos, éste se guarda como un nuevo slot en la tabla hash sólo si al insertarlo no se supera el máximo tamaño de la tabla, parámetro que es dado por la aplicación de usuario. La necesidad de este parámetro es no comprometer la memoria del cliente de la base de datos. Cuando la tabla hash llega a su tamaño máximo, es necesario quitar un elemento de ella para guardar el nuevo documento de metadatos. La elección del elemento a eliminar no es arbitraria, sino que se elige aquel elemento menos recientemente usado, siguiendo la convención del algoritmo de caché LRU (*Least Recent Used*). Esto implica que cuando se guarda un documento en la tabla hash, se le asocia una marca de tiempo (timestamp) que indica su último uso. Este valor se actualiza siempre que el elemento sea accedido. En el ejemplo de la figura 4.5, si el máximo tamaño de la caché es 5 y se necesita insertar un nuevo elemento, entonces se elimina el slot con la clave *fabricante* pues el documento de metadatos tiene el menor valor de timestamp de todos los valores de la tabla.

El tipo de metadatos para restricción de campos obligatorios que se introdujo en la sección 4.2.3 se guarda permanentemente en el caché porque siempre es consultado cada vez que se inserta un nuevo documento. De esta manera, no se registra el timestamp de este documento de metadatos. En la interfaz de integridad, la clase `Cache` es una subclase de `Hastable` que implementa el algoritmo LRU necesario para liberar el documento de metadatos menos recientemente usado cuando el caché llegó a su máximo.

El método `remove()` de la clase `Cache` es una implementación especial del algoritmo LRU. El valor `MAX_VALUE` es una constante del lenguaje de programación cuyo valor es el máximo posible para el tipo de dato. Cada elemento *e* es un par clave-valor de la tabla hash. El algoritmo elimina la clave cuyo documento asociado tiene el menor timestamp de todos, de modo que es el menos recientemente usado. El método `remove()` solo se invoca cuando el caché llega a su máximo.

```
void remove()
	min = MAX_VALUE
	for(Element e: cache)
		key = e.getKey()
		value = e.getValue()
		minTimestamp = value.getTimestamp()
		if(minTimestamp < min) then
			min = minTimestamp
			minKey = key
	cache.delete(minKey)
```

4.5.4. El algoritmo principal optimizado

El caché de metadatos es una estructura compartida por todos los hilos que procesan la validación de los documentos a insertar o actualizar. Esto hace necesario especificar mecanismos de sincronización que permitan mantener el caché de metadatos consistente. Es decir, se necesitan los medios para impedir que en el caché se eliminen o escriban concurrentemente los metadatos.
JAVA ofrece una implementación de semáforos, entendidos como enteros no negativos a los cuales se puede aplicar una de dos operaciones: *P(S)* y *V(S*), donde *S* es un semáforo [32]. Ambas operaciones son atómicas o indivisibles, es decir, en un momento dado, sólo un hilo puede ejecutar *P(S)* o *V(S)* [32]. El objetivo general de estas primitivas de sincronización es controlar el acceso a variables o estructuras de datos compartidas; en este caso, el caché de metadatos. Sea *S* un semáforo binario inicializado en 1. La primitiva *P*(*S)* disminuye el semáforo sólo si el resultado es no negativo. De otro modo el hilo que ejecuta *P(S)* espera hasta que *S = 1* [32]. La primitiva *V(S)* incrementa el semáforo en una unidad [32]. La idea intuitiva detrás del uso de semáforos es delimitar una sección de código (*sección crítica*) que accede a variables compartidas usando *P(S),* seguido de dicha sección de código, seguido de *V(S)*. Puede entenderse que al ejecutar *P(S),* el hilo que exitosamente entra a la sección crítica, impide que cualquier otro hilo ejecute dicha sección. Cuando el hilo termina de ejecutar esta sección, entonces ejecuta *V(S),* que significa que los recursos compartidos ahora están disponibles para cualquier otro hilo. A continuación se describe el algoritmo optimizado de verificación de inserción de documentos, considerando la implementación del caché de metadatos y el paralelismo. Se omiten nuevamente detalles irrelevantes de implementación.

Se asumen como variables globales:

- `cache`. La tabla hash que guarda los documentos de metadatos.
- `max`. Tamaño máximo del caché definido por la aplicación.
- `S`. Semáforo binario inicializado en 1.

```
String insertDoc(collection, document)
	correct = true
	correctAll = true
	for(Element e: document)
		field = e.getField()
		value = e.getValue()
		P(S)
		metadata = cache.get(field)
		V(S)
		if(metadata is null) then
			metadata = db.query("metadata": field)
				if(metadata not null) then
					metadata.timestamp = currentTime
					if(cache.size > max-1) then
						P(S)
						cache.remove()
						cache.put(metadata)
						V(S)
		if(metadata not null) then
			metadata.timestamp = currentTime
			if(metadata.containsField("set") then
				correct = validateSET(metadata, value)
			else
				correct = validate(metadata, value)
			correctAll = correctAll AND correct
	if(correctAll = true) then
		db.collection.insert(document)
		return "document inserted"
	else
		return "ERROR in document"
```

En el algoritmo, las líneas de código `metadata = cache.get(field)`, `cache.remove()` y `cache.put(metadata)` están delimitadas por las primitivas *P(S)* y *V(S),* lo cual logra exitosamente la sincronización de los hilos. En ningún momento dos hilos pueden eliminar o insertar concurrentemente un slot en el caché y en ningún momento un hilo lee datos del caché mientras otro hace una eliminación o inserción. De esta manera se preserva la consistencia del caché y se evita la ocurrencia de errores como metadatos duplicados, estados incorrectos, timestamps imprecisos o intentos por leer metadatos que ya no existen.

El algoritmo de actualización con verificación se basa en exactamente los mismos principios. Los cambios respecto del algoritmo para inserción son mínimos y tan sólo es diferente en el tipo de operación que se aplica en la base de datos.

4.5.5. Metadatos locales

Cuando no existen restricciones para ciertos atributos, la interfaz de integridad guarda en el caché de metadatos una lista de todos aquellos atributos que no requieren verificación.
Esto no sólo ahorra procesamiento, sino costosas consultas a la base de datos.
Este tipo de metadatos son locales en el sentido de que solo se guardan en la memoria principal del cliente que contiene la interfaz de integridad. Además, estos metadatos son permanentes en el caché, de modo que no es necesario almacenar su marca de tiempo.

4.6. Resumen

En este capítulo se han propuesto las estrategias que permiten implementar restricciones de integridad de dominio en una base de datos documental, utilizando una convención para la representación de metadatos y un programa intermedio (interfaz de integridad) entre la aplicación y dicha base de datos.
La aproximación considera características imprescindibles e importantes de optimización que incluyen indexación, paralelismo y cacheo de metadatos. Estas características se consideran como los medios necesarios para cubrir el requerimiento no funcional planteado en la hipótesis: el buen desempeño en el manejo de restricciones de integridad.
Las contribuciones de esta tesis, descritas en este capítulo, se llevan a la práctica en el capítulo 5, cuyo objetivo es la comprobación experimental de la hipótesis.

Capítulo 5. Casos de estudio

El objetivo de este capítulo es comprobar experimentalmente que el manejo de restricciones de integridad en una base de datos documental es una tarea cuyo tiempo de respuesta es directamente proporcional al conjunto de documentos a verificar.
Esto es, la estrategia propuesta en el Capítulo 4 se aplica a dos bases de datos y se mide el desempeño en ambos casos. Aquí se describe el proceso experimental y se muestran los resultados de esos experimentos.
La justificación de introducir datos de bases de datos relacionales (TPC) a un DBMS documental tiene que ver con la ausencia de un benchmark estandarizado para NoSQL. Además, las bases de datos de TPC se utilizan ampliamente en la industria del desarrollo de software.

5.1. La base de datos de TPC-E

Transaction Processing Performance Council (TPC) es una organización sin fines de lucro establecida en 1988 y fundada para definir el procesamiento de transacciones y benchmarks de bases de datos y para diseminar objetivamente el desempeño de datos para la industria [34]. TPC define *transacción* como un conjunto de operaciones que incluyen lectura y escritura a disco, llamadas a sistema operativo o cualquier forma de transferencia de datos de un subsistema a otro. Normalmente una transacción incluye actualizaciones en un sistema de base de datos [34]. TPC produce benchmarks que miden el procesamiento de transacciones y el desempeño de bases de datos en términos de cuántas transacciones puede procesar un sistema y una base de datos por unidad de tiempo [34]. Las empresas de la industria del software utilizan los benchmarks de TPC para ilustrar el desempeño competitivo de sus productos, y para mejorar y monitorear el desempeño de sus productos bajo desarrollo [34].

El benchmark TPC-E simula el trabajo transaccional de una empresa de intermediarios. Se enfoca en una base de datos central que ejecuta transacciones relacionadas a las cuentas de los clientes. El esquema de la base de datos, transacciones y reglas de implementación han sido diseñadas para ser ampliamente representativas de sistemas transaccionales modernos [22]. TPC-E modela la actividad de una empresa de intermediarios que debe administrar cuentas de clientes, ejecutar pedidos de intercambio, y responsabilizarse de las interacciones de los clientes con mercados financieros [22].
El siguiente diagrama ilustra el flujo de transacciones de un modelo de negocio retratado en el benchmark.

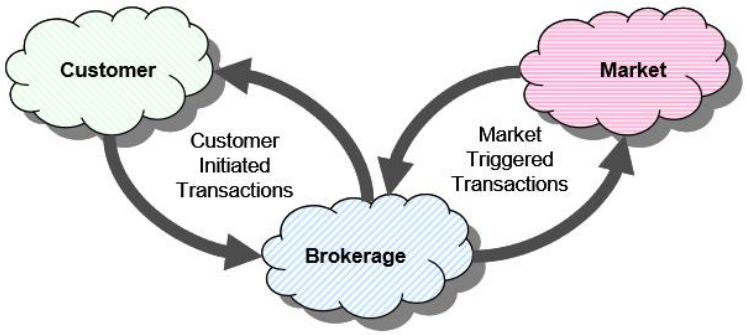

Figura 5.1. Flujo de transacciones en el modelo de negocio [22].

La compañía retratada por el benchmark tiene clientes que generan transacciones relacionadas a intercambios, solicitudes de cuentas e investigación de mercado. La empresa interactúa con mercados financieros para ejecutar pedidos de los clientes y actualizar información relevante en las cuentas [22]. El benchmark es escalable en el sentido de que el número de clientes definidos para la empresa de intermediarios puede variarse para representar las cargas de trabajo de diferentes negocios [22].

TPC-E está compuesto por un conjunto de transacciones que se ejecutan contra tres conjuntos de tablas en una base de datos que representan datos de mercado, datos de clientes y datos de agentes [22].

La figura 5.2 ilustra los componentes clave del entorno:

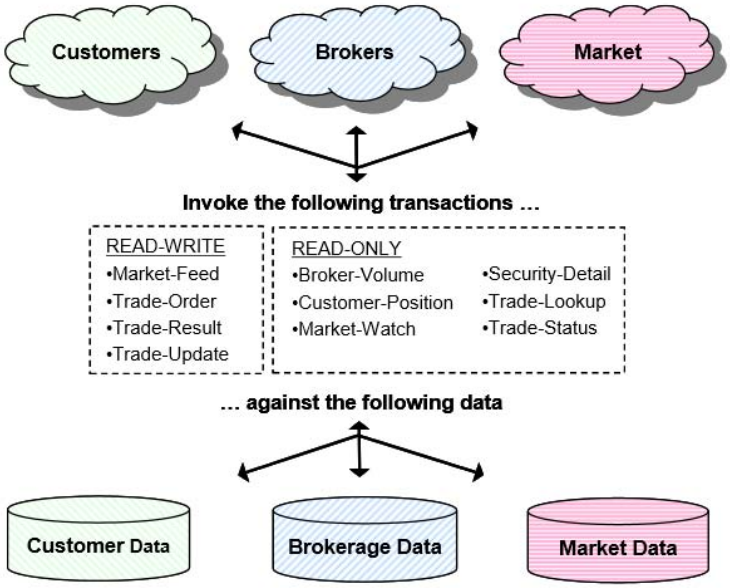

Figura 5.2. Componentes de la aplicación [22].

La base de datos TPC-E consiste de 33 tablas separadas e individuales. El esquema de la base de datos está organizado en cuatro conjuntos de tablas [22]:

- Tablas de cliente: Incluye 9 tablas que contienen información acerca de los clientes en la empresa de intermediarios.
- Tabla de agente: Incluye 9 tablas que contienen información acerca de la empresa de intermediarios y datos relacionados a agentes.
- Tablas de mercado: Incluye 11 tablas que contienen información acerca de compañías, mercados, intercambios y sectores de la industria.
- Tablas de dimensión: Incluye 4 tablas de dimensión que contienen información como direcciones y códigos postales.

5.2. La base de datos de TPC-H

El TPC Benchmark H es un benchmark de soporte a la decisión (DSS). Consiste de una suite de consultas ad-hoc orientadas al negocio y modificaciones de datos concurrentes. Las consultas y los datos que llenan la base de datos se han elegido para tener relevancia amplia en la industria [23]. El benchmark ilustra los sistemas de soporte a la decisión que examinan grandes volúmenes de datos, ejecutan consultas de un alto grado de complejidad y dan respuestas a preguntas críticas del negocio [23].
TPC-H no representa la actividad de un sector específico de la industria, sino para cualquier industria en donde se vende, administra o distribuye producto de forma masiva. Su propósito es reducir la diversidad de operaciones encontradas en una aplicación de análisis de información, aun manteniendo características esenciales de desempeño de la aplicación, es decir, el nivel de utilización de un sistema y la complejidad de operaciones [23].
La figura 5.3 ilustra el entorno de negocio de TPC-H y muestra las diferencias básicas con otros benchmarks.

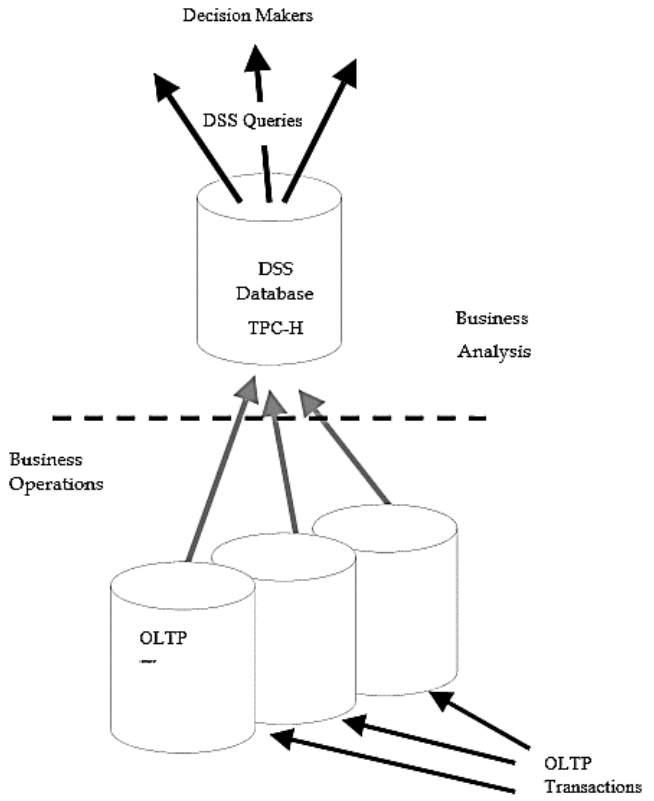

Figura 5.3. El entorno de negocio de TPC-H [23].

El benchmark TPC-H modela la parte de análisis de un entorno de negocio donde los datos son refinados y producidos para soportar la toma de decisiones de un negocio. En los benchmarks transaccionales los datos viajan hacia la base de datos desde varias fuentes donde es mantenida por un periodo de tiempo. En TPC-H, el contenido de una base de datos DSS es consultado por gente que toma decisiones en un negocio [23].

Los componentes de la base de datos de TPC-H consisten de 8 tablas separadas e individuales. Las relaciones entre columnas de estas tablas se ilustran en la figura 5.4 [23].

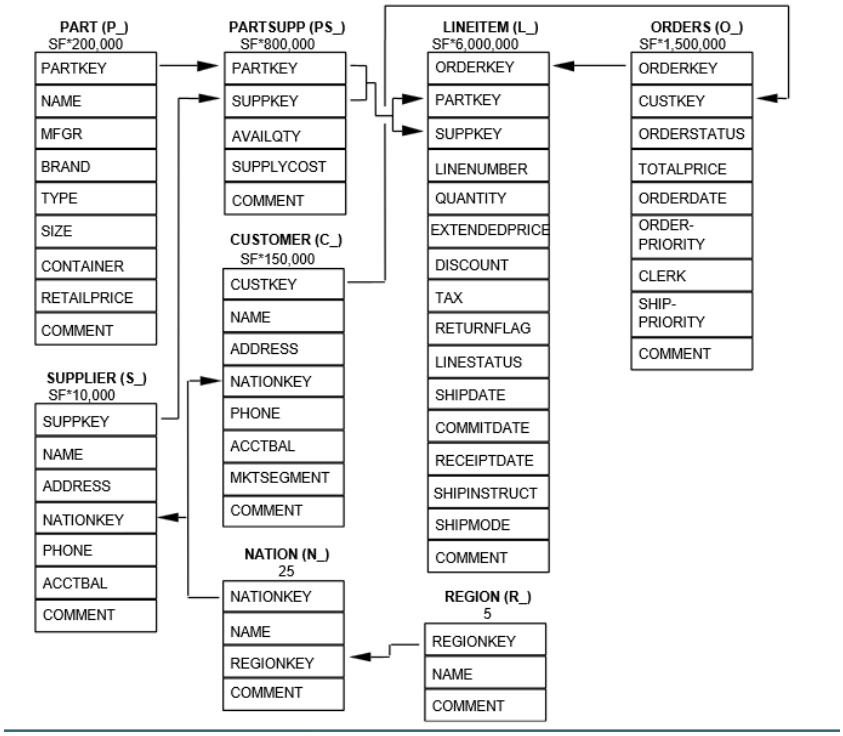

Figura 5.4. Esquema de la base de datos de TPC-H [23].

5.3. Diseño e implementación del experimento.

El objetivo del experimento es evaluar el desempeño de la interfaz de integridad utilizando los datos de los benchmarks TPC. Para lograr este objetivo se utilizan tablas específicas y selectas de las bases de datos de TPC, las cuales permitan especificar restricciones de integridad. De este modo no se pretende diseñar ni representar todo el esquema de TPC-E ni de TPC-H en MongoDB. Cada tabla seleccionada para el experimento se representa como una colección de documentos en MongoDB. Cada conjunto de restricciones en un atributo de una tabla es un documento de metadatos para dicha colección.

Las restricciones para las bases de datos descritas en los documentos de especificación de los benchmarks TCP-E y TPC-H se modifican o amplían con el fin de explotar lo mejor posible las funciones de la interfaz de integridad y las convenciones para la representación de metadatos establecidas en el Capítulo 4.

Adicionalmente, se omiten las restricciones de claves primarias y foráneas (incluyendo columnas de claves foráneas), pues dichas reglas están fuera del alcance de la funcionalidad de la interfaz de integridad, la cual se concentra exclusivamente en integridad de dominio.

En los apéndices A y B se encuentra la documentación acerca de las tablas y restricciones utilizadas para los benchmarks de TPC-E y TPC-H, respectivamente.

5.3.1. Metodología de pruebas de desempeño de la interfaz de integridad

El proceso de pruebas de desempeño requiere que se satisfagan antes los siguientes requerimientos.

Requerimientos

- **Asegurar que la interfaz de integridad es lo suficientemente estable.** Esto implica un proceso de depuración y pruebas de funcionalidad en una máquina local. Estas pruebas deben asegurar la correcta representación e inserción de los metadatos en MongoDB; y la correcta validación de documentos al momento de hacer inserciones y actualizaciones. Adicionalmente se optimiza código.

- **Obtención de datos de prueba de alta calidad**. Los benchmarks estandarizados de TPC proveen de herramientas para la generación automática de datos sintéticos. Estas herramientas son DBGEN (TPC-H) y EGEN (TPC-E). Estos programas facilitan la creación de archivos de texto separados por coma y de tamaños especificados por el usuario, lo cual permite obtener datos de prueba de distintos factores de escalamiento. Es decir, para cada tabla en el modelo de datos de los benchmarks pueden generarse archivos de tamaño diferente.

- **Configuración del ambiente.** La interfaz de integridad se compila en el equipo cliente de la base de datos. El cliente también debe contener el driver de MongoDB con las librerías que requiere la interfaz de integridad. Además, todos los archivos necesarios generados por DBGEN y EGEN deben residir en el cliente, pues son las entradas de la interfaz de integridad. El servidor debe ejecutar una instancia de MongoDB.

Entorno para la ejecución de pruebas de desempeño

El ambiente sobre el cual se ejecutan las pruebas de desempeño son dos nodos de un clúster: uno contiene la interfaz de integridad (cliente) y el otro ejecuta la instancia del servidor de MongoDB. Las especificaciones técnicas de los equipos son las siguientes:

Servidor
Procesador: 6 núcleos a 2.3 GHz
Capacidad de memoria RAM: 32 GB
Capacidad del disco: 5 TB

Cliente
Procesador: 12 núcleos a 2.5 GHz
Capacidad de memoria RAM : 64 GB
Capacidad del disco: 5 TB

El proceso de pruebas de desempeño

Una vez satisfechos los requerimientos, toda prueba de desempeño ejecuta los siguientes pasos:

- Por cada tabla seleccionada de TPC-H o TPC-E:
    1. La interfaz de integridad lee el archivo de texto que corresponde con esa tabla y con el factor de escalamiento especificado.
    2. La interfaz de integridad inserta metadatos en MongoDB que contienen restricciones específicas para los registros de ese archivo.
    3. Se recuperan los metadatos de MongoDB y se guardan en el caché de la interfaz de integridad.
    4. Cada registro del archivo es comparado con los metadatos correspondientes, y solo si el registro es correcto, éste es formateado a JSON y enviado a la base de datos MongoDB.
    5. Al término del procesamiento del archivo, la interfaz de integridad reporta el número de documentos insertados y el tiempo de respuesta. Éste tiempo se mide a partir de que la interfaz de integridad recupera los metadatos hasta el momento en que se terminó de procesar el último registro del archivo de texto. Los documentos insertados son los correctos considerando las restricciones, el resto son desechados.
    6. Repetir diez veces los pasos 1-5 por cada factor de escalamiento de la tabla seleccionada.
    7. Generar un reporte de los resultados para la tabla seleccionada. Se incluye una tabla cuyas columnas son el total de registros en el archivo, el número total de documentos insertados y el tiempo de respuesta. Cada fila en la tabla corresponde con uno de los factores de escalamiento usados.
    8. Generar una gráfica del factor de escalamiento contra tiempo de respuesta.
    9. Se analizan los resultados.

Se enfatiza que la redundancia de los pasos 2 y 3 es para simular una ejecución real del funcionamiento de la interfaz de integridad.
El número de documentos insertados siempre es menor que el número total de registros en el archivo, pues la interfaz desecha aquellos que no son correctos de acuerdo a las restricciones.

5.3.2. Comprobación del funcionamiento correcto de la interfaz de integridad

Si una base de datos documental se encuentra íntegra (de acuerdo a la definición dada en el Capítulo 4) luego de hacer cualquier número de escrituras, esto significa que la interfaz de integridad ha cumplido exitosamente con su función. Esto se verifica mediante la ejecución de consultas cuyo fin sea obtener documentos no válidos de la base de datos. Es decir, aquéllos que la interfaz debió filtrar de acuerdo a los metadatos previamente almacenados. El resultado de ejecutar cualquier consulta de este tipo debe ser vacío.
Por ejemplo, considere la tabla PART del benchmark TPC-H con las restricciones especificadas en el apéndice B. Entonces, cualquier consulta que busque documentos donde el campo P_PARTKEY sea menor que 100, 000 o P_SIZE sea menor que 25 no debe regresar ningún documento. A continuación se muestra la ejecución de estas consultas en la tabla

PART almacenada en MongoDB y llenada por la interfaz de integridad, en base a las restricciones definidas para esta tabla en el apéndice B. Las consultas no recuperan ningún documento, como se esperaba.

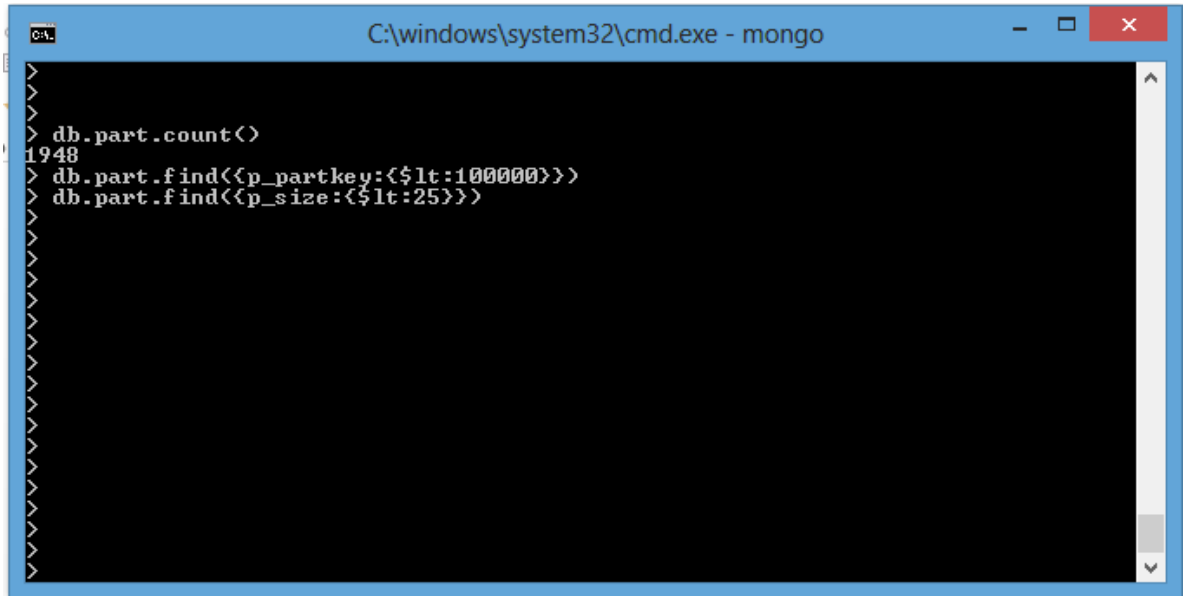

Figura 5.5. Comprobación del comportamiento correcto de la interfaz de integridad mediante consultas.

Durante el proceso de pruebas de desempeño se muestran evidencias por cada restricción de cómo la interfaz de integridad asegura la inserción de sólo los documentos correctos. En dichas pruebas esto se documenta siguiendo la convención descrita.

5.4. Resultados de desempeño

Esta sección presenta los resultados de desempeño en términos de tiempo de respuesta al momento de ejecutar la interfaz de integridad utilizando diferentes volúmenes de información de los datos sintéticos de los benchmarks TPC-E y TPC-H. En cada experimento se describe la configuración de la interfaz de integridad, el estado de la base de datos y las tablas utilizadas. Todos los tiempos de respuesta son promedios de diez ejecuciones. La dispersión de los tiempos se obtiene usando la siguiente fórmula:

$$t = M \pm (2.5) * \frac{S}{\sqrt{n-1}}$$

Donde *M* es la media, *S* es la varianza, *n* es el número de ejecuciones y 2.5 es una constante que se utiliza para la distribución de *student* que tiene un 97.5% de confianza.

Todos los experimentos funcionan con 50 hilos de ejecución, pues se obtienen resultados óptimos con tal configuración. Es decir, dicha cantidad de hilos permite cumplir con la meta de desempeño establecida.

En ningún experimento el tamaño máximo del archivo de entrada llega a 1 GB, pues al llegar a ese límite se generan cuellos de botella y excepciones en tiempo de ejecución. Esto es debido a una excesiva cantidad de procesamiento ocupada por hilos del recolector de basura de JAVA.

5.4.1. Resultados de los experimentos con datos de TPC-E

La base de datos del benchmark TPC-E tiene tablas que crecen en función de la cantidad de filas de la tabla CUSTOMER especificadas. En los siguientes experimentos se utilizan archivos cuyo tamaño cambia de acuerdo a las cantidades de 1000, 2000, 3000 y 4000 registros de CUSTOMER. Por simplicidad, dichas cantidades son referidas como factores de escalamiento 1, 2, 3 y 4, respectivamente; lo cual refiere a archivos de un tamaño fijo y la creación de otros tres archivos del doble, triple y cuádruple tamaño, respectivamente.
En la columna de número de registros se muestra entre paréntesis el factor de escalamiento.

1) **Estado de la base de datos**: Contiene solo los metadatos.
   Configuración de la interfaz de integridad:
   Cantidad de documentos de metadatos: 5.
   Capacidad del caché: 10.
   Escrituras: 50 hilos insertan.
   Tablas utilizadas: CUSTOMER_ACCOUNT.

   El experimento 1 tiene como objetivo comprobar el comportamiento óptimo de la interfaz de integridad, en el caso del procesamiento de una pequeña cantidad de datos. En esta configuración las transacciones solo insertan documentos.

Resultados

De los resultados en la tabla 5.1 y la figura 5.6 se observa que al trabajar con un pequeño volumen de datos las diferencias de tiempo de respuesta no son observables. Para esta situación ni siquiera existe dispersión entre los tiempos de respuesta dentro de un mismo factor de escalamiento. Lo anterior se atribuye totalmente a la poca cantidad de documentos validados, lo que genera menos tráfico en la red y una menor cantidad de accesos a la base de datos.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
5000 (1)	1 ± 0	63
10000 (2)	1 ± 0	149
15000 (3)	1 ± 0	247
20000 (4)	1 ± 0	355

Tabla 5.1. Resultados de desempeño del experimento 1.

Gráfica del tiempo promedio de respuesta

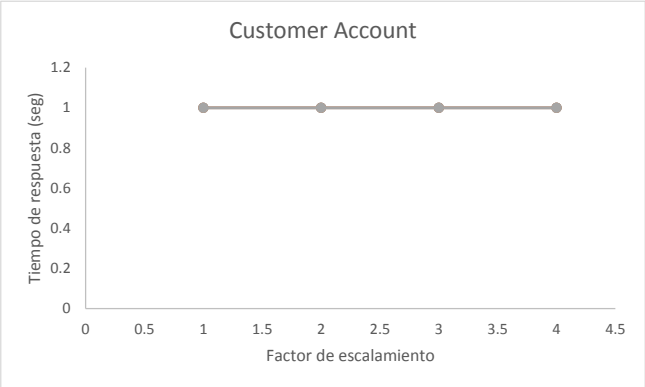

Figura 5.6. Gráfica de tiempo promedio del experimento 1.

Comprobación del funcionamiento correcto de la interfaz de integridad

La figura 5.7 muestra consultas a la colección CUSTOMER_ACCOUNT (factor de escalamiento 1), en donde se buscan documentos que no deben existir, pues fueron desechados por la interfaz de integridad con base en los metadatos. Ninguna de las consultas recupera documentos, como se esperaba. Esto comprueba el comportamiento correcto de la interfaz de integridad.

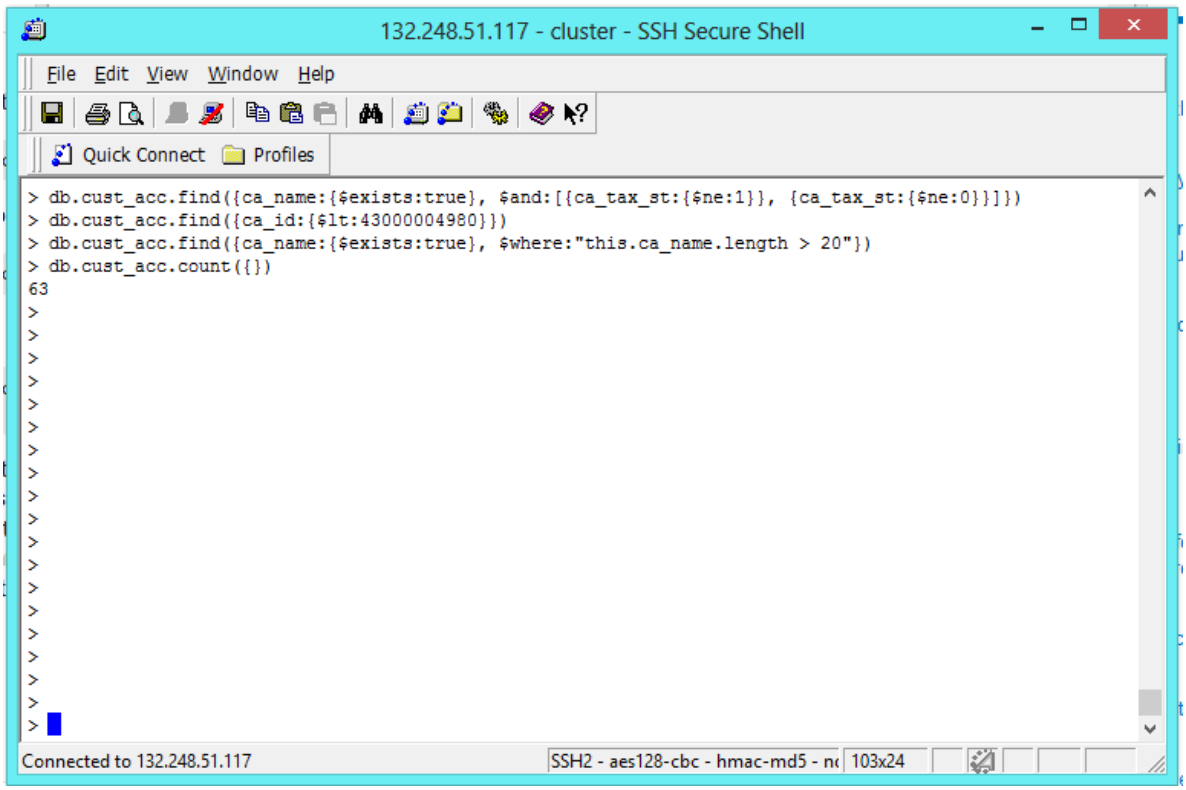

Figura 5.7. Comprobación del funcionamiento de la interfaz de integridad.

2) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 5.
Capacidad del caché: 10.
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: CUSTOMER_ACCOUNT.

El experimento 2 presenta una configuración parecida al experimento 1, y busca encontrar contrastes (si se presentan), en el caso de sustituir 10 hilos de inserción por actualizaciones.

Resultados

Los resultados de la tabla 5.2 y la figura 5.8 muestran un comportamiento similar que en el experimento 1. La explicación es que los hilos de actualización utilizan exactamente el mismo algoritmo de validación de documentos. Se observa que tampoco existen retardos de red o latencia de acceso al disco duro del servidor.

Número de registros en el archivo	Tiempo de respuesta promedio (seg)	Documentos insertados
5000 (1)	1 ± 0	35
10000 (2)	1 ± 0	114
15000 (3)	1 ± 0	194
20000 (4)	1 ± 0	262

Tabla 5.2. Resultados de desempeño del experimento 2.

Gráfica del tiempo promedio de respuesta

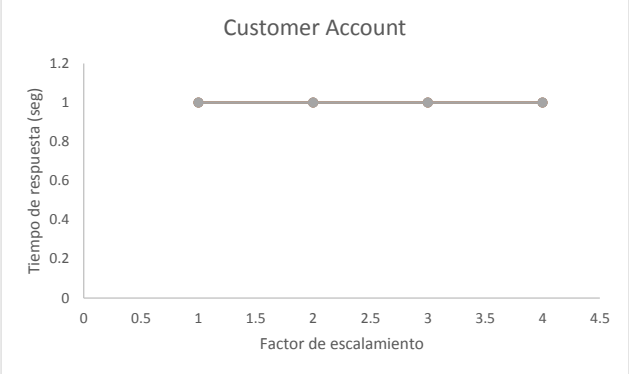

Figura 5.8. Gráfica de tiempo promedio del experimento 2.

3) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 7
Capacidad del caché: 10
Escrituras: 50 hilos insertan.
Tablas utilizadas: DAILY MARKET.

El experimento 3 tiene como objetivo observar el desempeño de la interfaz de integridad, en el caso del procesamiento de varias decenas de megabytes.

Resultados

Los resultados de la tabla 5.3 y la figura 5.9 muestran que se tiene mayor dispersión en los tiempos de respuesta. Esto se debe a factores como inherentes retardos de la red y latencia de acceso del servidor a su disco duro. No obstante, se observa la tendencia a un crecimiento directamente proporcional al volumen de documentos a validar.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
893925 (1)	93.1 ± 2.48	24899
1787850 (2)	191.5 ± 23.93	50865
2681775 (3)	279.2 ± 92.36	76501
3575700 (4)	369.4 ± 49.47	102775

Tabla 5.3. Resultados de desempeño del experimento 3.

Gráfica del tiempo promedio de respuesta

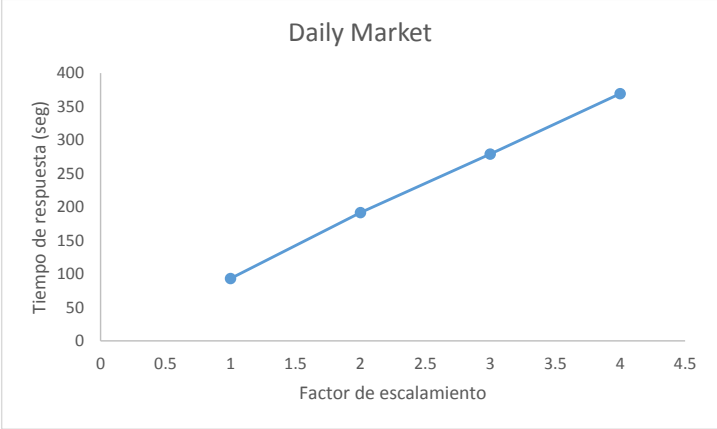

Figura 5.9. Gráfica de tiempo promedio del experimento 3.

Comprobación del funcionamiento correcto de la interfaz de integridad

La figura 5.10 muestra consultas a la colección DAILY_MARKET (factor de escalamiento 1), en donde se buscan documentos que no deben existir, pues fueron desechados por la interfaz de integridad con base en los metadatos. Ninguna de las consultas recupera documentos, como se esperaba. Esto comprueba el comportamiento correcto de la interfaz de integridad.

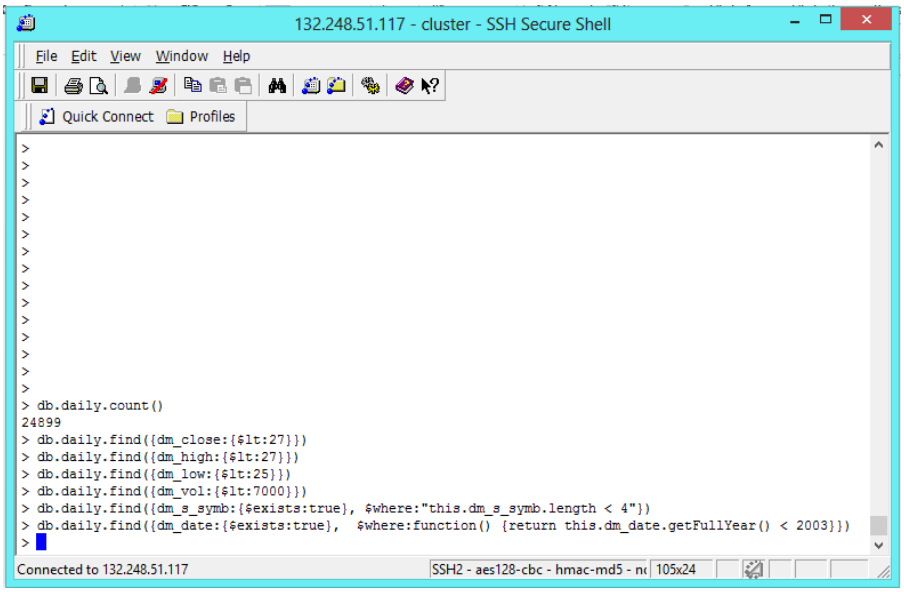

Figura 5.10. Comprobación del funcionamiento de la interfaz de integridad.

4) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 7
Capacidad del caché: 10
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: DAILY MARKET.

El experimento 4 presenta una configuración parecida al experimento 3, y busca encontrar contrastes (si se presentan), en el caso de sustituir 10 hilos de inserción por actualizaciones.

Resultados

Los resultados de la tabla 5.4 y la figura 5.11 indican que aun sustituyendo 10 hilos de inserción por actualización, el tiempo de respuesta tiende a crecer de forma directamente proporcional al volumen de datos a procesar. Esto ocurre debido a que el algoritmo de validación es el mismo para inserción y para actualización. El experimento también sugiere que al tener un menor volumen de documentos a insertar existe una menor dispersión de los tiempos de respuesta para un mismo factor de escalamiento.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
893925 (1)	67.2 ± 3.65	19948
1787850 (2)	142.9 ± 21.74	40342
2681775 (3)	206.5 ± 31.34	61338
3575700 (4)	281.6 ± 54.37	81382

Tabla 5.4. Resultados de desempeño del experimento 4.

Gráfica del tiempo promedio de respuesta

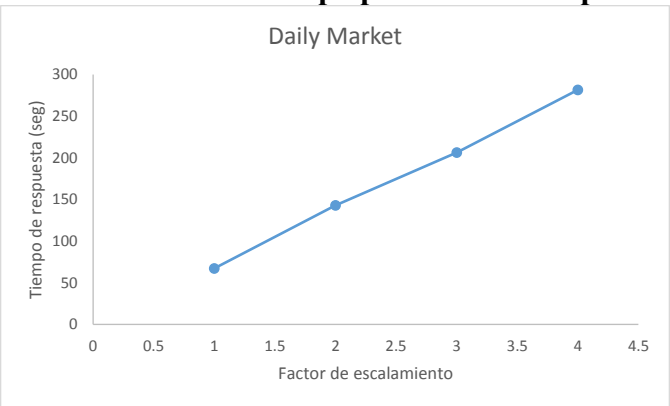

Figura 5.11. Gráfica de tiempo promedio del experimento 4.

5) **Estado de la base de datos**: Contiene solo los metadatos indexados.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 11
Capacidad del caché: 7
Escrituras: 50 hilos insertan.
Tablas utilizadas: DAILY MARKET y CUSTOMER ACCOUNT.

El experimento 5 incrementa la carga de datos respecto de los experimentos 3 y 4. Para esto se han procesado registros de los archivos de dos tablas diferentes.
Además, la configuración simula la situación en la que no todos los documentos de metadatos caben en el caché. Se busca observar el comportamiento de la interfaz de integridad en dicha circunstancia.

Resultados

Los resultados de la tabla 5.5 y la figura 5.12 sugieren que cuando no caben en el caché todos los documentos de metadatos, el tiempo de respuesta deja de crecer de forma proporcional al volumen de documentos y tiene un comportamiento incierto. Esto sucede porque los hilos requieren hacer consultas a la base de datos por cada metadato de cada campo que no reside en el caché. Dicha cantidad de consultas tiende a aumentar a varias centenas de miles o hasta millones, y es muy costosa porque introduce retardos de la red y latencia de acceso en el servidor a su disco duro. Otro hecho derivado del experimento es que aun usando la indexación, no se resuelve el cuello de botella de las consultas para la obtención de metadatos.

Número de registros en el archivo (DAILY MARKET)	Número de registros en el archivo (CUSTOMER ACCOUNT)	Tiempo de respuesta promedio (seg)	Documentos insertados
893925 (1)	5000 (1)	53.2 ± 19.4	24962
1787850 (2)	10000 (2)	232.5 ± 28.75	51014
2681775 (3)	15000 (3)	414.8 ± 35.78	76748
3575700 (4)	20000 (4)	594 ± 223.3	103130

Tabla 5.5. Resultados de desempeño del experimento 5.

Gráfica del tiempo promedio de respuesta

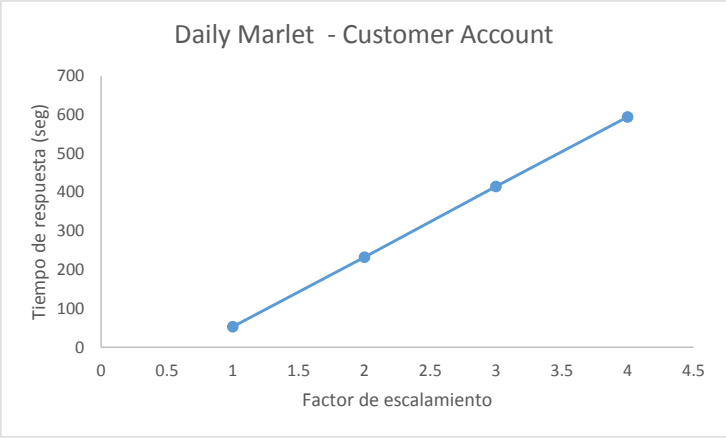

Figura 5.12. Gráfica de tiempo promedio del experimento 5.

6) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 11
Capacidad del caché: 15
Escrituras: 50 hilos insertan.
Tablas utilizadas: DAILY MARKET y CUSTOMER ACCOUNT.

El experimento 6 presenta una configuración similar al experimento 5, con la diferencia en que todos los metadatos caben en el caché. Se pretende observar si dicha situación favorece, y en qué grado, el desempeño de la interfaz de integridad.

Resultados

La máquina cliente accede directamente a todos los metadatos y por cada lectura a la memoria principal tan sólo gasta algunos nanosegundos, lo cual no tiene ningún impacto negativo en el tiempo total de respuesta. Los resultados de la tabla 5.6 y la figura 5.13 muestran que el tiempo de respuesta crece directamente proporcional al volumen de documentos a validar. De este experimento se concluye que el almacenamiento de todos los metadatos en el caché resuelve el cuello de botella y mejora en gran medida, el desempeño de la interfaz de integridad.

Número de registros en el archivo (DAILY MARKET)	Número de registros en el archivo (CUSTOMER ACCOUNT)	Tiempo de respuesta promedio (seg)	Documentos insertados
893925 (1)	5000 (1)	84.7 ± 4.26	24962
1787850 (2)	10000 (2)	169.7 ± 12.78	51014
2681775 (3)	15000 (3)	247 ± 15.5	76748
3575700 (4)	20000 (4)	331.5 ± 39.3	103130

Tabla 5.6. Resultados de desempeño del experimento 6.

Gráfica del tiempo promedio de respuesta

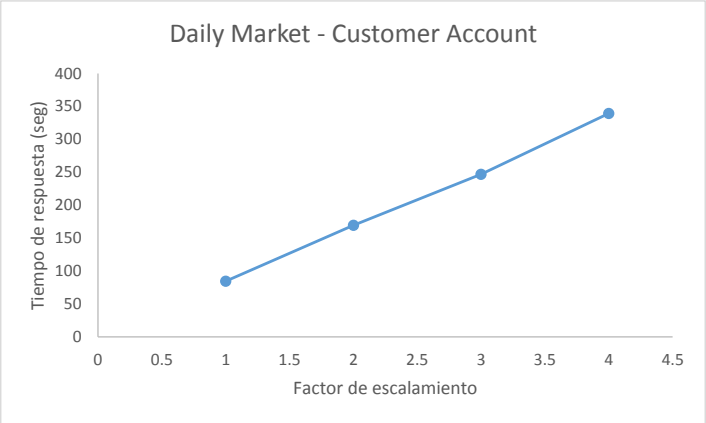

Figura 5.13. Gráfica de tiempo promedio del experimento 6.

7) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 8
Capacidad del caché: 5
Escrituras: 50 hilos insertan.
Tablas utilizadas: COMPANY, LAST TRADE.

El experimento 7 utiliza archivos con pocos registros, pero sus restricciones son diferentes a aquellas del experimento 1. Además, el caché solo guarda algunos metadatos. Se pretende observar el desempeño de la interfaz de integridad ante esta situación.

Resultados

Los resultados de la tabla 5.7 y la figura 5.14 sugieren que cuando el volumen de documentos a procesar es muy pequeño, es irrelevante si todos los metadatos se encuentran en el caché porque la cantidad de consultas a la base de datos para recuperar metadatos crece muy poco y no impacta el tiempo de respuesta. La dispersión de los tiempos de respuesta para el mismo factor de escalamiento es nula. Se observa que debido a las restricciones implementadas, en cada prueba tan sólo se insertan los documentos de metadatos.

Número de registros en el archivo (COMPANY)	Número de registros en el archivo (LAST TRADE)	Tiempo de respuesta promedio (seg)	Documentos insertados
500 (1)	685 (1)	1 ± 0	8
1000 (2)	1370 (2)	1 ± 0	8
1500 (3)	2055 (3)	1 ± 0	8
2000 (4)	2740 (4)	1 ± 0	8

Tabla 5.7. Resultados de desempeño del experimento 7.

Gráfica del tiempo promedio de respuesta

Figura 5.14. Gráfica de tiempo promedio del experimento 7.

Comprobación del funcionamiento correcto de la interfaz de integridad

La figura 5.15 muestra consultas a la colección COMPANY_LAST_TRADE (factor de escalamiento 1), en donde se buscan documentos que no deben existir, pues fueron desechados por la interfaz de integridad con base en los metadatos. En dichas consultas se comprueba la correcta implementación de las restricciones de ambas tablas: COMPANY y LAST_TRADE. Ninguna de las consultas recupera documentos, como se esperaba. Esto comprueba el comportamiento correcto de la interfaz de integridad.

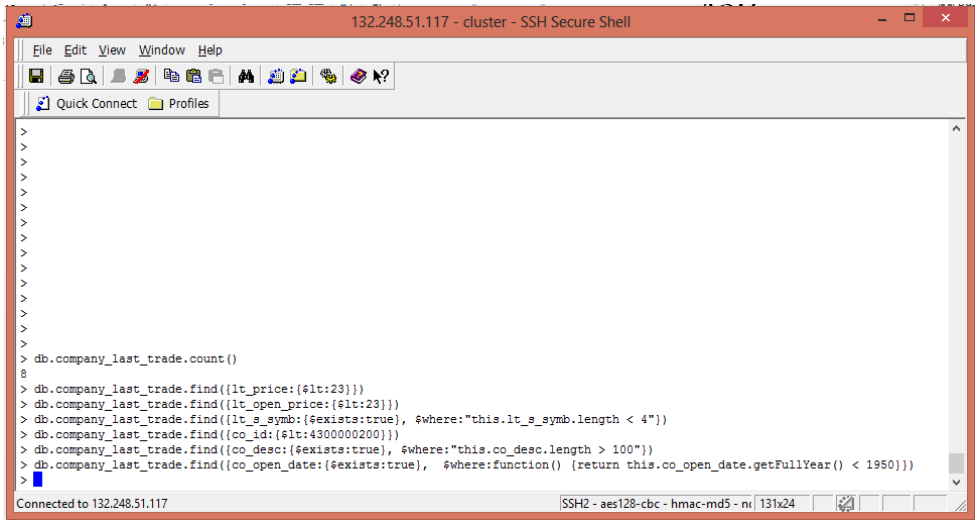

Figura 5.15. Comprobación del funcionamiento de la interfaz de integridad.

8) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 8
Capacidad del caché: 5
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: COMPANY, LAST TRADE.

La configuración es similar a la del experimento 7, excepto que se han reemplazado 10 hilos de inserción por actualización.

Resultados

La tabla 5.8 y la figura 5.16 indican que se han obtenido resultados similares al experimento 7, y esto sugiere que las actualizaciones no impactan el crecimiento proporcional del tiempo de respuesta, sino que se sigue manteniendo una constante independiente de pequeños incrementos. Se observa que debido a las restricciones implementadas, en cada prueba tan sólo se insertan los documentos de metadatos.

Número de registros en el archivo (COMPANY)	**Número de registros en el archivo (LAST TRADE)**	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
500 (1)	685 (1)	1 ± 0	8
1000 (2)	1370 (2)	1 ± 0	8
1500 (3)	2055 (3)	1 ± 0	8
2000 (4)	2740 (4)	1 ± 0	8

Tabla 5.8. Resultados de desempeño del experimento 8.

Gráfica del tiempo promedio de respuesta

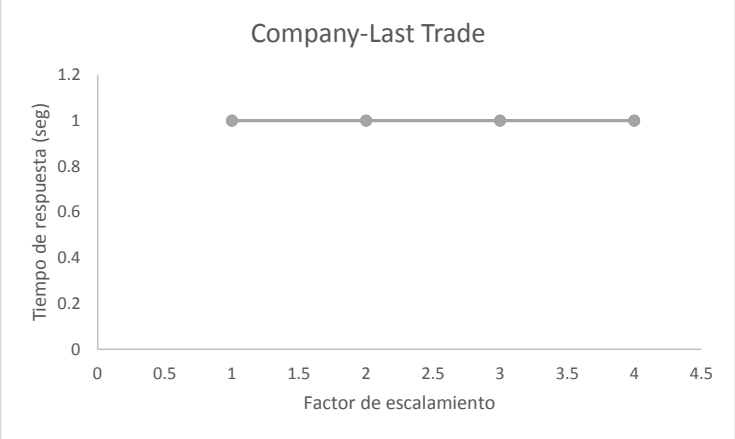

Figura 5.16. Gráfica de tiempo promedio del experimento 8.

A continuación se repiten algunas de las pruebas anteriormente formuladas pero utilizando otros factores de escalamiento que permite el benchmark de TPC-E, con la finalidad de descubrir y analizar cómo se comporta el tiempo de respuesta. Los experimentos 9 y 10 utilizan los factores de escalamiento 2, 4, 6 y 8 mientras que los siguientes dos experimentos utilizan los factores de 1, 2, 4 y 8.

9) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 7
Capacidad del caché: 10
Escrituras: 50 hilos insertan.
Tablas utilizadas: DAILY MARKET.

Resultados

El cambio por otros factores de escalamiento tan sólo ha afectado la dispersión que existe en los tiempos de respuesta para diferentes ejecuciones de una misma transacción. Sin embargo, de la tabla 5.9 y la figura 5.17 se observa que la tendencia es que el tiempo de respuesta crece directamente proporcional conforme aumenta el volumen de datos a procesar.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
1787850 (2)	192.1 ± 2.3	50865
3575700 (4)	370.7 ± 46.85	102775
5363550 (6)	577 ± 112.5	154382
7151400 (8)	789.1 ± 184.16	207697

Tabla 5.9. Resultados de desempeño del experimento 9.

Gráfica del tiempo promedio de respuesta

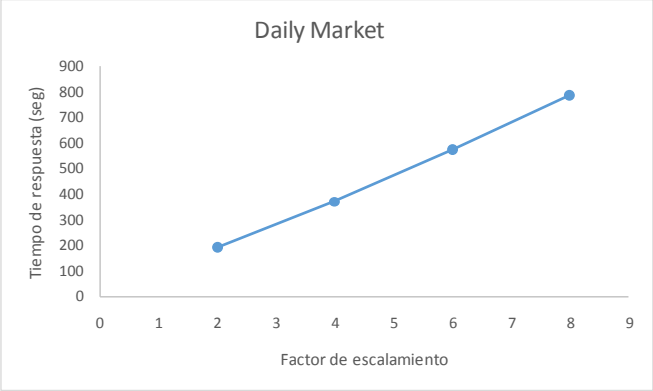

Figura 5.17. Gráfica de tiempo promedio del experimento 9.

10) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 7
Capacidad del caché: 10
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: DAILY MARKET.

Resultados

Aun reemplazando hilos de inserción por actualización el tiempo de respuesta tiende a crecer de manera proporcional, independientemente del cambio por otros factores de escalamiento. La dispersión de los tiempos de respuesta también tiende a crecer cuando aumenta la cantidad de datos. La tabla 5.10 y la figura 5.18 muestran los resultados del experimento.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
1787850 (2)	147.7 ± 39.63	40342
3575700 (4)	288.3 ± 31.67	81382
5363550 (6)	439.5 ± 37.26	123699
7151400 (8)	629.5 ± 263.37	165010

Tabla 5.10. Resultados de desempeño del experimento 10.

Gráfica del tiempo promedio de respuesta

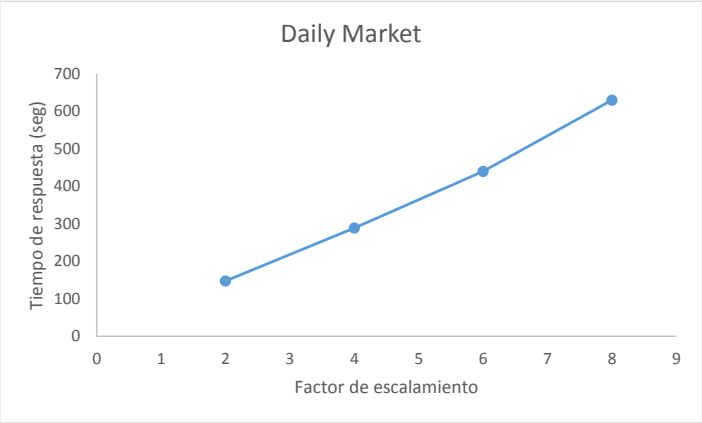

Figura 5.18. Gráfica de tiempo promedio del experimento 10.

11) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 11
Capacidad del caché: 15
Escrituras: 50 hilos insertan.
Tablas utilizadas: DAILY MARKET y CUSTOMER ACCOUNT.

Resultados

Los resultados muestran que el tiempo de respuesta crece de manera proporcional a la cantidad de datos. Esto es cierto aun utilizando otros factores de escalamiento. La dispersión de los tiempos promedios tiende a aumentar conforme crece el volumen de datos. La tabla 5.11 y la figura 5.19 muestran los resultados del experimento.

Número de registros en el archivo (DAILY MARKET)	**Número de registros en el archivo (CUSTOMER ACCOUNT)**	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
893925 (1)	5000 (1)	90.3 ± 2.2	24962
1787850 (2)	10000 (2)	171.2 ± 11.99	51014
3575700 (4)	20000 (4)	337.7 ± 49.63	103130
7151400 (8)	40000 (8)	723.2 ± 59.59	207700

Tabla 5.11. Resultados de desempeño del experimento 11.

Gráfica del tiempo promedio de respuesta

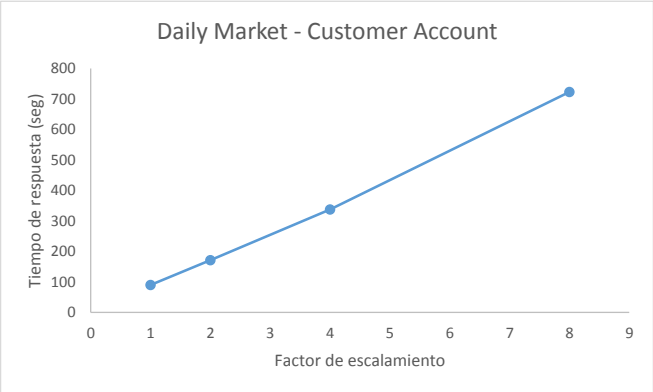

Figura 5.19. Gráfica de tiempo promedio del experimento 11.

12) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 11
Capacidad del caché: 15
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: DAILY MARKET y CUSTOMER ACCOUNT.

Resultados

Los resultados de la tabla 5.12 y la figura 5.20 confirman lo establecido para el experimento 11 aún si se reemplazan 10 hilos de inserción por actualizaciones.

Número de registros en el archivo (DAILY MARKET)	**Número de registros en el archivo (CUSTOMER ACCOUNT)**	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
893925 (1)	5000 (1)	51.1 ± 13.04	15413
1787850 (2)	10000 (2)	100.6 ± 7.9	29793
3575700 (4)	20000 (4)	198.3 ± 31.11	61341
7151400 (8)	40000 (8)	427.1 ± 78.04	123702

Tabla 5.12. Resultados de desempeño del experimento 12.

Gráfica del tiempo promedio de respuesta

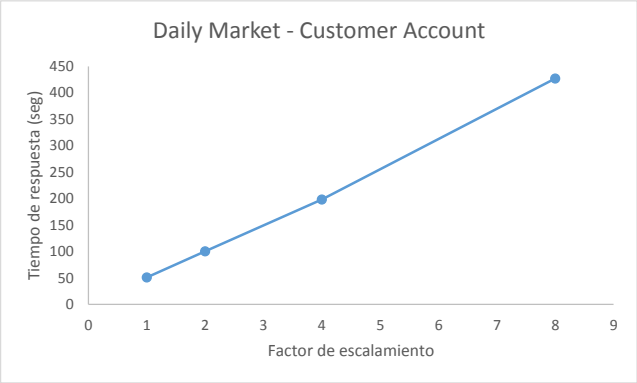

Figura 5.20. Gráfica de tiempo promedio del experimento 12.

5.4.2. Resultados de los experimentos con datos de TPC-H

Los factores de escalamiento utilizados para estos experimentos son de 1, 10 y 30, los cuales generan archivos de un volumen fijo y otros dos que tienen aproximadamente diez y treinta veces su tamaño, respectivamente. En la columna de tamaño se muestra entre paréntesis el factor de escalamiento.

1) **Estado de la base de datos**: Contiene solo los metadatos.
   Configuración de la interfaz de integridad:
   Cantidad de documentos de metadatos: 9.
   Capacidad del caché: 10.
   Escrituras: 50 hilos insertan.
   Tablas utilizadas: PART.

Resultados

Los resultados de la tabla 5.13 y la figura 5.21 muestran que el tiempo de respuesta crece directamente proporcional al volumen de documentos a validar. Sin embargo, la dispersión de los tiempos de respuesta tiene que ver con la desproporción en la cantidad de documentos insertados entre factores de escalamiento. Aunque el tamaño de las tablas crece en factores fijos, el volumen los documentos correctos e insertados no crece linealmente y esto impacta el desempeño. Es decir, existen inevitables costos de inserción.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
200000 (1)	31.5 ± 0.95	1948
2000000 (10)	282.4 ± 70.58	35300
6000000 (30)	799.5 ± 333.3	108639

Tabla 5.13. Resultados de desempeño del experimento 1.

Gráfica del tiempo promedio de respuesta

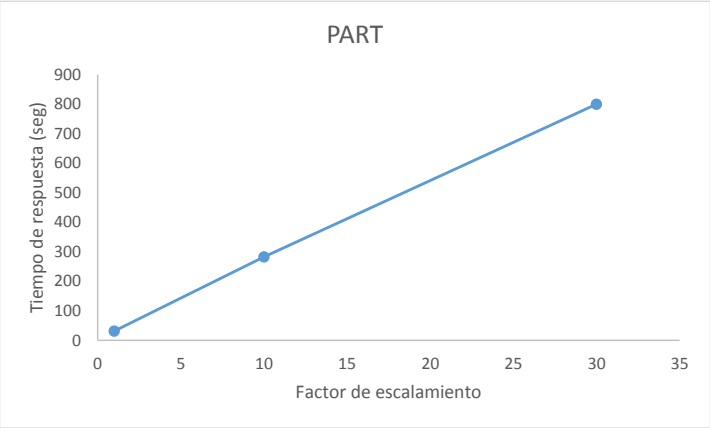

Figura 5.21. Gráfica de tiempo promedio del experimento 1.

Comprobación del funcionamiento correcto de la interfaz de integridad

La figura 5.22 muestra consultas a la colección PART (factor de escalamiento 1), en donde se buscan documentos que no deben existir, pues fueron desechados por la interfaz de integridad con base en los metadatos. Ninguna de las consultas recupera documentos, como se esperaba. Esto comprueba el comportamiento correcto de la interfaz de integridad.

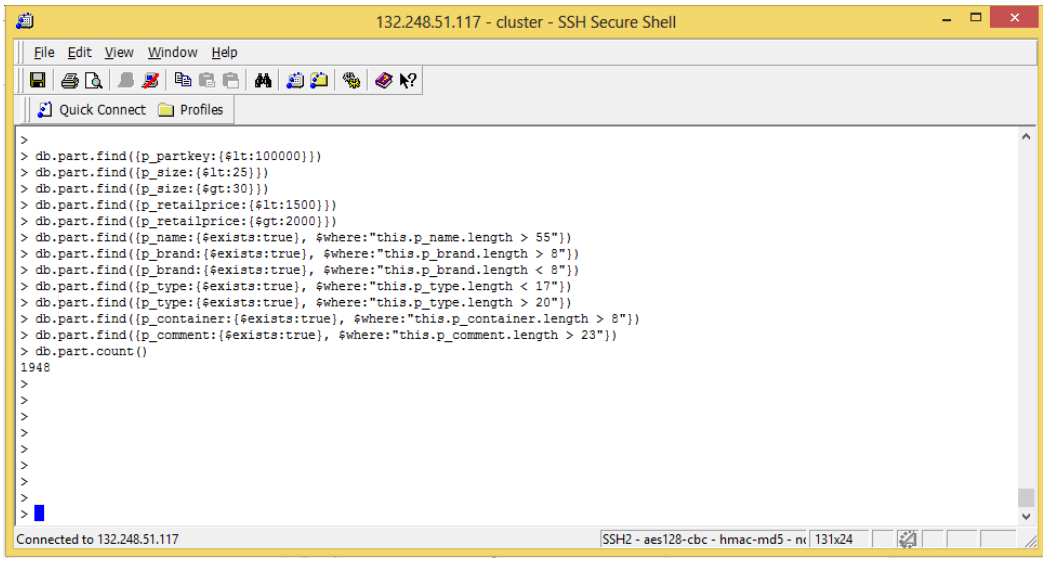

Figura 5.22. Comprobación del funcionamiento de la interfaz de integridad.

2) **Estado de la base de datos**: Contiene solo los metadatos.
   Configuración de la interfaz de integridad:
   Cantidad de documentos de metadatos: 9.
   Capacidad del caché: 10.
   Escrituras: 40 hilos insertan, 10 hilos actualizan.
   Tablas utilizadas: PART.

En el experimento 2 se han reemplazado 10 hilos de inserción por actualización, manteniendo iguales el resto de las condiciones del experimento 1.

Resultados

Los resultados de la tabla 5.14 y la figura 5.23 confirman lo establecido en el experimento 1. El contraste con el desempeño del primer experimento se debe principalmente a un crecimiento mucho menor en la cantidad de documentos a insertar y no tiene que ver con la sustitución de inserciones por actualizaciones.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
200000 (1)	14.7 ± 0.19	1168
2000000 (10)	212.3 ± 11.67	27845
6000000 (30)	621.7 ± 163.70	86587

Tabla 5.14. Resultados de desempeño del experimento 2.

Gráfica del tiempo promedio de respuesta

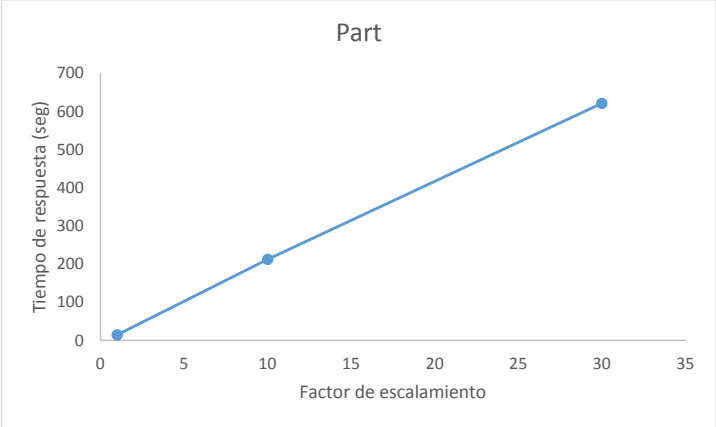

Figura 5.23. Gráfica de tiempo promedio del experimento 2.

3) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 7.
Capacidad del caché: 10.
Escrituras: 50 hilos insertan.
Tablas utilizadas: CUSTOMER.

Este experimento tiene como fin descubrir si existe una cantidad óptima de hilos. Es decir, aquella que minimiza el tiempo de respuesta requerido para el procesamiento de un conjunto de registros. De esta manera se muestran resultados de tiempos de respuesta para 30, 50 y 70 hilos de ejecución. El proceso de pruebas de desempeño es el mismo en todo el experimento.

Resultados

Los resultados muestran que la cantidad óptima de hilos es 30. La ganancia en tiempo de respuesta usando 30 hilos se obtiene solo hasta que se procesa un volumen de información moderadamente grande. También se observa que en los tres casos, el crecimiento del tiempo de respuesta es directamente proporcional a los datos de entrada. La tabla 5.15 y la figura 5.24 muestran los resultados del experimento.

Número de registros en el archivo	**Tiempo (seg) usando 50 threads**	**Tiempo (seg) usando 30 Threads**	**Tiempo (seg) usando 70 Threads**	**Documentos insertados**
150000 (1)	19.3 ± 0.74	18.4 ± 1.14	19.6 ± 0.59	24697
1500000 (10)	181.1 ± 16.19	168.2 ± 14.2	199.4 ± 37.8	465789
4500000 (30)	522.9 ± 104.52	509.3 ± 54.2	534.9 ± 101.5	1447736

Tabla 5.15. Resultados de desempeño del experimento 3.

Gráfica del tiempo promedio de respuesta

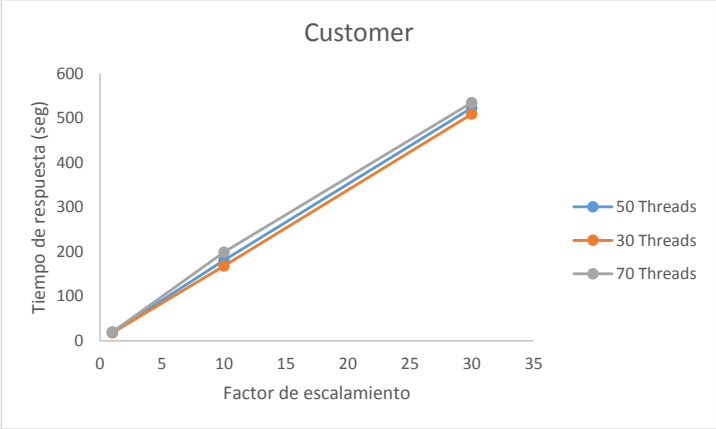

Figura 5.24. Gráfica de tiempo promedio del experimento 3.

Comprobación del funcionamiento correcto de la interfaz de integridad

La figura 5.25 muestra consultas a la colección CUSTOMER (factor de escalamiento 1), en donde se buscan documentos que no deben existir, pues fueron desechados por la interfaz de integridad con base en los metadatos. Ninguna de las consultas recupera documentos, como se esperaba. Esto comprueba el comportamiento correcto de la interfaz de integridad.

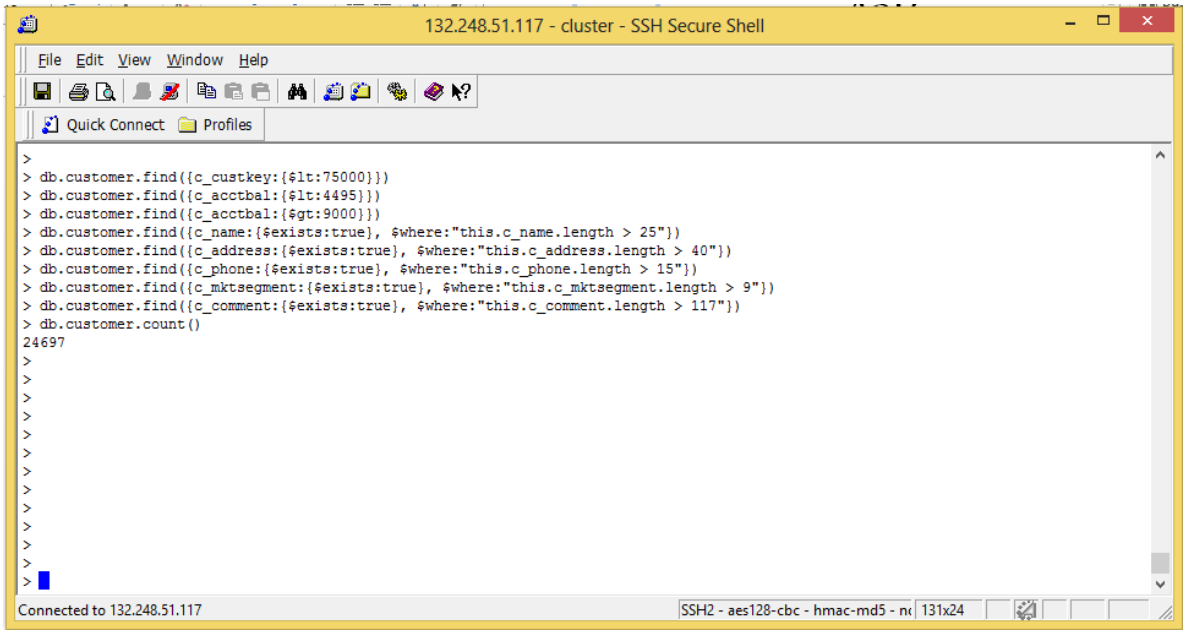

Figura 5.25. Comprobación del funcionamiento de la interfaz de integridad.

4) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 7.
Capacidad del caché: 10.
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: CUSTOMER.

Resultados

Los resultados de la tabla 5.16 y la figura 5.26 evidencian que es muy significativo el crecimiento de la cantidad de documentos que se insertan en la base de datos porque esto introduce un retardo adicional por la latencia de red y la escritura en el disco duro del servidor. El contraste con el desempeño del experimento 3 se atribuye a un crecimiento mucho menor en la cantidad de documentos a insertar.

Número de registros en el archivo	Tiempo de respuesta promedio (seg)	Documentos insertados
150000 (1)	10.5 ± 0.22	14618
1500000 (10)	140.9 ± 11.93	367662
4500000 (30)	409 ± 13.6	1153447

Tabla 5.16. Resultados de desempeño del experimento 4.

Gráfica del tiempo promedio de respuesta

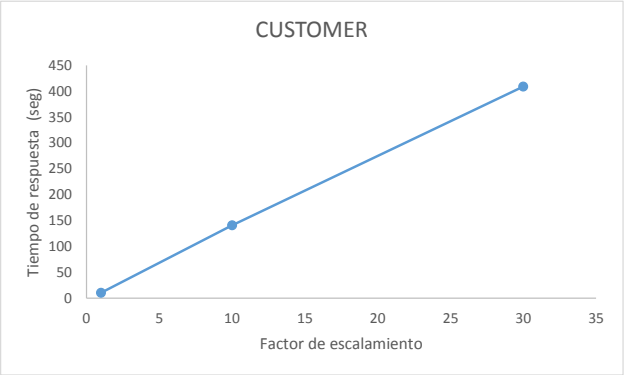

Figura 5.26. Gráfica de tiempo promedio del experimento 4.

5) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 6.
Capacidad del caché: 10.
Escrituras: 50 hilos insertan.
Tablas utilizadas: SUPPLIER.

Resultados

Los resultados de la tabla 5.17 y la figura 5.27 muestran que el tiempo de respuesta crece directamente proporcional al volumen de documentos a validar. La dispersión de los tiempos de respuesta entre una ejecución y otra parecen ser mucho menores cuando se trabaja con unas cuantas decenas de megabytes.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
10000 (1)	1 ± 0	2470
100000 (10)	10.5 ± 0.22	47324
300000 (30)	29.4 ± 1.3	147879

Tabla 5.17. Resultados de desempeño del experimento 5.

Gráfica del tiempo promedio de respuesta

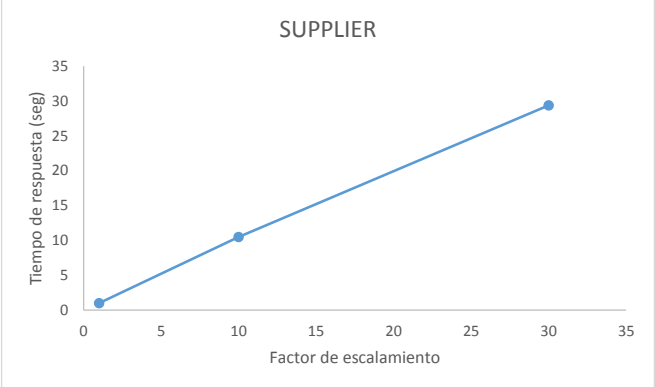

Figura 5.27. Gráfica de tiempo promedio del experimento 5.

Comprobación del funcionamiento correcto de la interfaz de integridad

La figura 5.28 muestra consultas a la colección SUPPLIER (factor de escalamiento 1), en donde se buscan documentos que no deben existir, pues fueron desechados por la interfaz de integridad con base en los metadatos. Ninguna de las consultas recupera documentos, como se esperaba. Esto comprueba el comportamiento correcto de la interfaz de integridad.

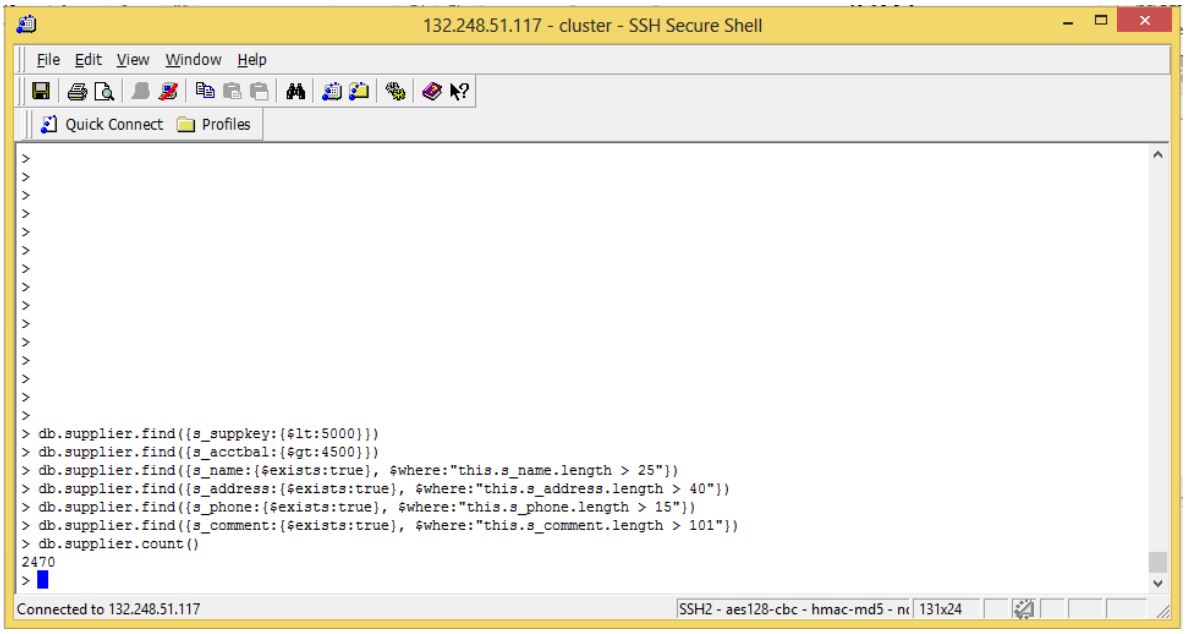

Figura 5.28. Comprobación del funcionamiento de la interfaz de integridad.

6) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 6.
Capacidad del caché: 10.
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: SUPPLIER.

Resultados

Los resultados de la tabla 5.18 y la figura 5.29 sugieren que el reemplazo de 10 hilos de inserción por actualizaciones no mueve la tendencia de crecimiento del tiempo de respuesta. Al contrario, las diferencias entre una ejecución y otra son menores. Esto es debido a que el algoritmo de actualización es el mismo, pero se transmiten menos datos a través de la red.

Número de registros en el archivo	Tiempo de respuesta promedio (seg)	Documentos insertados
10000 (1)	1 ± 0	1495
100000 (10)	6.2 ± 0.51	37317
300000 (30)	21.1 ± 1	117574

Tabla 5.18. Resultados de desempeño del experimento 6.

Gráfica del tiempo promedio de respuesta

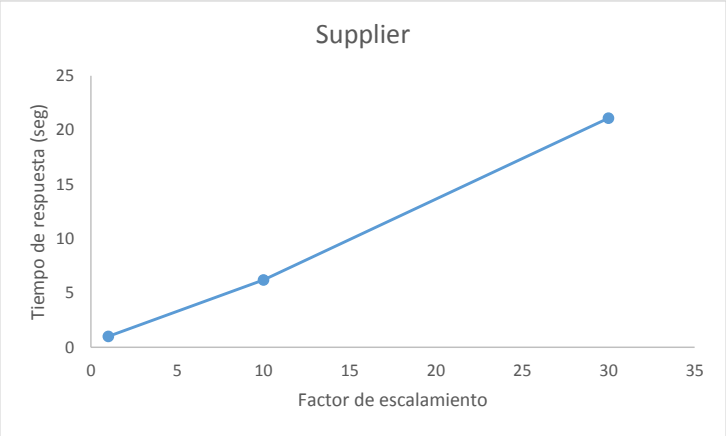

Figura 5.29. Gráfica de tiempo promedio del experimento 6.

7) **Estado de la base de datos**: Contiene solo los metadatos indexados.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 6.
Capacidad del caché: 3.
Escrituras: 50 hilos insertan.
Tablas utilizadas: SUPPLIER.

Resultados

Los resultados de la tabla 5.19 y la figura 5.30 sugieren que cuando todos los documentos de metadatos no caben en el caché, el tiempo de respuesta deja de crecer de forma proporcional al volumen de documentos y tiene un comportamiento incierto. Esto sucede porque los hilos requieren hacer consultas a la base de datos por cada metadato de cada campo que no reside en el caché. Dicha cantidad de consultas tiende a aumentar a varias centenas de miles o hasta millones y es muy costosa porque introduce retardos de la red y latencia de acceso en el servidor a su disco duro. Adicionalmente, el volumen de documentos insertados no crece de manera proporcional y esto introduce más retardos en el servidor.
Otro hecho derivado del experimento es que aun usando la indexación no se resuelve el cuello de botella de las consultas para la obtención de metadatos.
Todo lo anterior es cierto aun procesando unas pocas decenas de megabytes.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
10000 (1)	1 ± 0	2470
100000 (10)	27.9 ± 0.07	47324
300000 (30)	90.1 ± 0.63	147879

Tabla 5.19. Resultados de desempeño del experimento 7.

Gráfica del tiempo promedio de respuesta

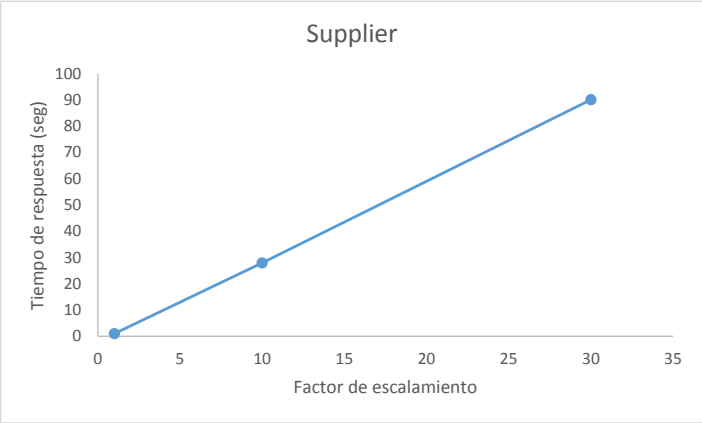

Figura 5.30. Gráfica de tiempo promedio del experimento 7.

8) **Estado de la base de datos**: Contiene solo los metadatos indexados.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 6.
Capacidad del caché: 3.
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: SUPPLIER.

Resultados

Este experimento confirma todo lo establecido para el experimento 7, aún y cuando se reemplacen 10 hilos de inserción por actualización. Los resultados se muestran en la tabla 5.20 y la figura 5.31.

Número de registros en el archivo	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
10000 (1)	1 ± 0	1495
100000 (10)	22 ± 0.18	37317
300000 (30)	71.1 ± 0.08	117574

Tabla 5.20. Resultados de desempeño del experimento 8.

Gráfica del tiempo promedio de respuesta

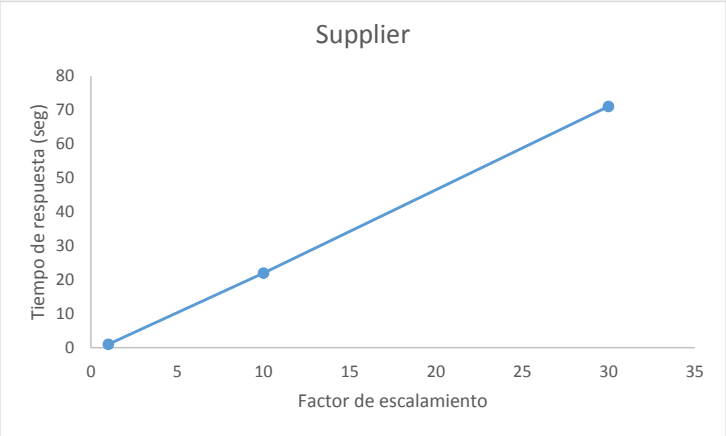

Figura 5.31. Gráfica de tiempo promedio del experimento 8.

9) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 13
Capacidad del caché: 15
Escrituras: 50 hilos insertan.
Tablas utilizadas: CUSTOMER y SUPPLIER.

Resultados

Los resultados de la tabla 5.21 y la figura 5.32 muestran que el tiempo de respuesta crece directamente proporcional al volumen de documentos a validar. La dispersión de los tiempos de respuesta tiene que ver con inherentes retardos de la red y latencia de acceso en el servidor a su disco duro.

Número de registros en el archivo (CUSTOMER)	**Número de registros en el archivo (SUPPLIER)**	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
150000 (1)	10000 (1)	20.8 ± 0.516	27167
1500000 (10)	100000 (10)	198.7 ± 68.89	513113
4500000 (30)	300000 (30)	554.2 ± 128.10	1,595,615

Tabla 5.21. Resultados de desempeño del experimento 9.

Gráfica del tiempo promedio de respuesta

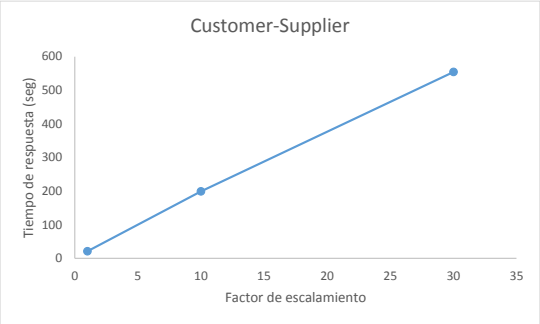

Figura 5.32. Gráfica de tiempo promedio del experimento 9.

10) **Estado de la base de datos**: Contiene solo los metadatos.
Configuración de la interfaz de integridad:
Cantidad de documentos de metadatos: 13
Capacidad del caché: 15
Escrituras: 40 hilos insertan, 10 hilos actualizan.
Tablas utilizadas: CUSTOMER y SUPPLIER.

Resultados

Los resultados de la tabla 5.22 y la figura 5.33 indican que el reemplazo de 10 hilos de inserción por actualizaciones no mueve la tendencia de crecimiento del tiempo de respuesta. Al contrario, las diferencias entre una ejecución y otra son porque se transmiten menos documentos a través de la red y se escriben menos datos en la base de datos.

Número de registros en el archivo (CUSTOMER)	**Número de registros en el archivo (SUPPLIER)**	**Tiempo de respuesta promedio (seg)**	**Documentos insertados**
150000 (1)	10000 (1)	16.7 ± 0.19	25203
1500000 (10)	100000 (10)	178.1 ± 13.78	493211
4500000 (30)	300000 (30)	529 ± 23.08	1,535,118

Tabla 5.22. Resultados de desempeño del experimento 10.

Gráfica del tiempo promedio de respuesta

Figura 5.33. Gráfica de tiempo promedio del experimento 10.

5.5. Resumen

Este capítulo ilustra mediante dos casos de estudio, la aplicación de la propuesta introducida en el Capítulo 4 para el control de integridad de datos en un sistema manejador de base de datos documental. Los datos de los benchmarks estándares TPC-E y TPC-H se utilizaron con este fin y se diseñaron metadatos para algunas de sus tablas. Al ejecutar diferentes pruebas con algunos de los factores de escalamiento que los benchmarks proveen, se han obtenido resultados de desempeño que indican que el tiempo de respuesta para validar documentos es una tarea cuyo tiempo de respuesta crece directamente proporcional al volumen de datos a procesar. Esto se cumple siempre que la capacidad del caché sea lo suficientemente grande para almacenar todos los metadatos.

En el capítulo 6 se realiza un análisis detallado y global de cada caso de estudio, documentando y considerando las observaciones de todos los experimentos con datos de TPC-H y TPC-E.

Capítulo 6. Análisis y conclusiones

6.1. Resumen del trabajo

En este trabajo de tesis se aportan las estrategias necesarias para la implementación eficiente de restricciones de integridad en una base de datos NoSQL documental.
Las estrategias comprenden los algoritmos necesarios de validación de información, la especificación de un modelo de metadatos y las características de optimización que sirven para cumplir el objetivo de obtener un buen desempeño al momento de validar documentos.

6.2. Re-enunciado de la hipótesis

Los análisis de las secciones 6.3, 6.4 y las conclusiones responden de manera detallada y en base a la experimentación a la pregunta planteada en la hipótesis:

Dada una base de datos documental, ¿Es posible implementar externamente el procesamiento de restricciones de integridad de dominio de tal forma que la validación de los documentos sea una tarea cuyo tiempo de respuesta crezca directamente proporcional a la cantidad de documentos a verificar?

6.3. Análisis de los resultados de TPC-E

En general, cuando el caché de metadatos tiene capacidad suficiente, existe una relación directamente proporcional entre el volumen de datos a procesar y el tiempo de respuesta. Los resultados muestran que la excepción a este comportamiento se presenta cuando se procesan pequeños volúmenes de datos, es decir, aquéllos que se encuentran alrededor de 1 MB. En esos casos el tiempo de respuesta se mantiene como una constante independientemente de los incrementos. En dichas circunstancias, cuando la capacidad del caché es insuficiente, eso no tiene ningún efecto negativo en el desempeño, y el tiempo de respuesta aún sigue siendo una constante independiente de los incrementos. Lo anterior no es cierto para decenas de megabytes en adelante, pues existe un cuello de botella debido a la necesidad de hacer demasiadas consultas a la base de datos por cada metadato requerido que no se encuentra en el caché. Además, dicha cantidad de consultas de metadatos no crece linealmente y menos su tiempo de respuesta.
La dispersión de los tiempos de respuesta puede ser afectada por factores como los retardos de la red y el tiempo de acceso en el servidor a su disco duro.
La utilización de otros factores de escalamiento que permite el benchmark lleva a las mismas conclusiones y no se observan diferencias sustanciales en el comportamiento del tiempo de respuesta.

6.4. Análisis de los resultados de TPC-H

Los resultados del benchmark TPC-H establecen que la aplicación de actualizaciones en sustitución de inserciones regularmente decrece el tiempo de respuesta. El algoritmo de validación de actualizaciones es exactamente el mismo que se utiliza para inserciones, como se mencionó en el Capítulo 4. De modo que cada hilo de actualización se limita a validar y enviar sólo un documento, no un conjunto de documentos. Esto causa que se envíen menos datos a través de la red y disminuye el tiempo total de respuesta.
La otra característica importante de desempeño que revelan estos resultados es la influencia tan importante que tiene la cantidad de documentos insertados sobre el tiempo de respuesta total. En el caso de TPC-E, era una casualidad que con los metadatos especificados, la cantidad de documentos válidos y por lo tanto insertados creciera aproximadamente de forma lineal. En TPC-H esto no es cierto para algunas tablas, y el desempeño se ve severamente afectado por los inevitables costos de escritura en la base de datos. Es decir, el procesamiento total de documentos correctos e insertados crece de forma desproporcionada entre factores de escalamiento y esto impacta el tiempo de respuesta.
Uno de los experimentos sugiere que la cantidad de hilos generados por la interfaz de integridad es una característica que puede afectar el rendimiento. Esto es importante si interesa minimizar el tiempo de respuesta, pero los resultados muestran que en todos los casos se cumple la meta de desempeño.
Los resultados de TPC-H también confirman lo establecido en los resultados de TPC-E: el tiempo de respuesta tiene un crecimiento directamente proporcional respecto de la cantidad de datos siempre que el tamaño del caché sea suficiente para almacenar todos los metadatos. De otro modo la tendencia de crecimiento de los tiempos de respuesta es completamente incierta. Y esto se cumple aunque se utilice indexación.

6.5. Conclusiones

Los diferentes experimentos realizados con los benchmarks de TPC-E y TPC-H llevan a las siguientes conclusiones generales. El tiempo de respuesta de la interfaz de integridad es directamente proporcional a la cantidad de documentos a validar. Las dos excepciones a esta regla se producen cuando:

- Se trabaja con pequeñas cantidades de datos (alrededor de 1 MB), donde el tiempo de respuesta es una constante que se vuelve independiente de incrementos.
- El caché no contiene todos los metadatos y se están procesando al menos varias decenas de megabytes, en cuyo caso el crecimiento del tiempo de respuesta es incierto debido a una sobrecarga por demasiadas consultas a la base de datos para obtener metadatos. Esto es verdad aún y cuando se utilice indexación en la base de datos.

Vale completamente la pena reservar espacio en la memoria principal de la máquina cliente que contiene la aplicación y la interfaz de integridad para almacenar todos los documentos de metadatos. Aún en grandes bases de datos reales con cientos de tablas y todos los campos con restricciones, el caché apenas ocuparía algunos megabytes.
El desempeño de la interfaz de integridad no se ve afectado por solicitudes de actualización de la aplicación cliente porque el algoritmo de validación es exactamente el mismo que para

inserciones. Entonces la interfaz de integridad puede manejar adecuadamente cualquier tipo de escrituras en la base de datos.
La latencia de red y el tiempo de acceso del servidor a su disco duro son características que pueden afectar el desempeño de la interfaz de integridad.

6.6. Contribuciones

- Se provee de un marco de referencia para la representación de metadatos en bases de datos NoSQL documentales.
- Se aportan los algoritmos necesarios para la implementación de restricciones de integridad de dominio en un DBMS documental.
- Se describen en detalle las estrategias que permiten obtener el mejor desempeño en términos de tiempos de respuesta. Esto incluye indexación en la base de datos y paralelismo y cacheo de metadatos del lado del cliente.
- Finalmente existe un análisis de desempeño bajo el mismo criterio y con base en todos los experimentos realizados.

6.7. Trabajos futuros

El control de la integridad de la información es un problema general de todas las bases de datos NoSQL, y vale la pena estudiar cómo aplicar las estrategias descritas en este trabajo y adaptarlas a otros tipos de DBMS, como los basados en columnas, clave-valor y orientados a grafos. Además, en cada uno de esos tipos de bases de datos pueden existir otras optimizaciones que ayuden a minimizar los tiempos de respuesta.

Pueden hacerse pruebas de escalabilidad a la interfaz de integridad, con el fin de descubrir qué tan bien se adapta a diferentes plataformas. Incluso, puede estudiarse como hacer un balance de carga automatizado, en lugar de determinar manualmente la cantidad de hilos óptima.

Es conveniente la estandarización de un lenguaje de consultas para DBMS documentales, ya que facilitaría en gran medida el aprendizaje de los mismos. Además esto es factible pues la mayoría utilizan formatos comunes como JSON o XML.

Capítulo 7. Bibliografía y referencias

[1] http://sg.com.mx/revista/42/nosql-la-evolucion-las-bases-datos
[2] http://docs.mongodb.org/manual/core/crud-introduction/
[3] http://www.mongodb.com/nosql-explained
[4] Kashyap, Suman, Shruti Zamwar, Tanvi Bhavsar, and Snigdha Singh. *Benchmarking and Analysis of NoSQL Technologies*.
[5] Tudorica, Bodgan George, and Cristian Bucur. *A comparison between several NoSQL databases with comments and notes*. In Roedunet International Conference (RoEduNet), 2011 10th, pp. 1-5. IEEE, 2011.
[6] http://www.sitepen.com/blog/2010/05/11/nosql-architecture/
[7] *A Relational Model Of Data For Large Shared Databanks*, Codd E.F., 1970.
[8] *Fundamentos de sistemas de bases de datos*, Elmasri R, Navathe S.B., Pearson Addison Wesley, quinta edición, 2007, 998 pp.
[9] *Database Systems: The Complete Book,* Garcia H, Ullman J.D., Widom Jennifer, Pearson/Prentice Hall, segunda edición, 2009, 1203 pp.
[10] *SEQUEL: A Structured English Query Language,* Chamberlin Donald D, Boyce Raymond F, 1974.
[11] http://www.cavsi.com/preguntasrespuestas/que-es-un-sistema-gestor-de-bases-de-datos-o-sgbd/
[12] *Fundamentos y Modelos de Bases de Datos.*A. de Miguel y M. Piattini,Ed. 2. RA-MA. 1999.
[13] *NoSQL databases*, Strauch, C., Sites, U. L. S., & Kriha, W. (2011).
[14] Orend, Kai. *"Analysis and classification of NoSQL databases and evaluation of their ability to replace an object-relational Persistence Layer."* Architecture(2010), 100.
[15] http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html
[16] Leavitt, Neal. *Will NoSQL databases live up to their promise?*. Computer 43.2 (2010): 12-14.
[17] http://nosql-database.org/
[18] http://sg.com.mx/revista/42/nosql-la-evolucion-las-bases-datos#.UyZeM4Uadco
[19] http://gravitar.biz/bi/base-datos-columnar/
[20] http://nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf
[21] *Principles of distributed database systems*, Özsu M.T., Valduriez P, Springer, Tercera edición, 845 Pp.
[22] http://www.tpc.org/tpce/spec/v1.13.0/tpce-v1.13.0.pdf
[23] http://www.tpc.org/tpch/spec/tpch2.17.0.pdf
[24] *Referential Integrity in Cloud NoSQL Databases.* (2012), Raja, Harsha.
[25] *SQL Support over MongoDB using Metadata*, Khan, S., & Mane, V, International Journal of Scientific and Research Publications, Volume 3, Issue 10, 2013.
[26] *Referential integrity and dependencies between documents in a document oriented database*. Georgiev, Kalin, GSTF Journal on Computing 2, no. 4 (2013).
[27] *Database System Concepts,* Silberschatz, Korth, Sudarshan, McGraw-Hill, cuarta edición, 2004, 918 pp.
[28] http://docs.mongodb.org/manual/core/indexes-introduction/
[29] http://docs.mongodb.org/manual/core/index-compound/
[30] http://docs.mongodb.org/manual/core/data-model-operations/#data-model-indexes

[31] *Introduction to algorithms,* Cormen et al, The MIT Press, Tercera edición, 2009, 1292 Pp.
[32] Dijkstra, Edsger W. "Cooperating sequential processes." (1968): 43-112.
[33] Molyneaux, Ian. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. " O'Reilly Media, Inc.", 2009, 145 Pp.
[34] http://www.tpc.org/information/about/abouttpc.asp
[35] Tiwari, Shashank. *Professional NoSQL*. John Wiley & Sons, 2011.
[36] Moniruzzaman, A. B. M., and Syed Akhter Hossain. *"NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison*." arXiv preprint arXiv:1307.0191 (2013).

Apéndice A: Tablas y restricciones de TPC-E

Categoría: Clientes
Nombre de la tabla: CUSTOMER_ACCOUNT

Nombre de columna	Tipo de dato	Restricciones
CA_ID	Numérico	No nulo, > 43000004980
CA_NAME	Cadena	No nulo, máx: 20
CA_TAX_ST	Entero	No nulo, dominio: {0, 1}
CA_BAL	Numérico	No nulo

Tabla A-1. Restricciones para la tabla CUSTOMER_ACCOUNT de la base de datos TPC-E.

Categoría: Mercado
Nombre de la tabla: LAST_TRADE

Nombre de columna	Tipo de dato	Restricciones
LT_S_SYMB	Cadena	No nulo, longitud > 4
LT_DTS	Fecha	No nulo
LT_PRICE	Numérico	No nulo, > 23
LT_OPEN_PRICE	Numérico	No nulo, > 23
LT_VOL	Numérico	No nulo

Tabla A-2. Restricciones para la tabla LAST_TRADE de la base de datos TPC-E.

Categoría: Mercado
Nombre de la tabla: DAILY MARKET

Nombre de columna	Tipo de dato	Restricciones
DM_DATE	Fecha	No nulo, año >= 2003
DM_S_SYMB	Cadena	No nulo, longitud > 4
DM_CLOSE	Numérico	No nulo, > 27
DM_HIGH	Numérico	No nulo, > 27
DM_LOW	Numérico	No nulo, > 25
DM_VOL	Numérico	No nulo, > 7000

Tabla A-3. Restricciones para la tabla DAILY_MARKET de la base de datos TPC-E.

Categoría: Mercado
Nombre de la tabla: COMPANY

Nombre de columna	Tipo de dato	Restricciones
CO_ID	Numérico	No nulo, > 4300000200
CO_NAME	Cadena	No nulo
CO_SP_RATE	Cadena	No nulo
CO_CEO	Cadena	No nulo
CO_DESC	Cadena	No nulo, max: 100
CO_OPEN_DATE	Fecha	No nulo, año > 1950

Tabla A-4. Restricciones para la tabla COMPANY de la base de datos TPC-E.

Apéndice B: Tablas y restricciones de TPC-H

Nombre de la tabla: PART

Nombre de columna	Tipo de dato	Restricciones
P_PARTKEY	Numérico	>= 100, 000
P_NAME	Cadena	<= 55
P_MFGR	Cadena	N/A
P_BRAND	Cadena	Texto fijo de 8 caracteres
P_TYPE	Cadena	Longitud: > 17, < 20
P_SIZE	Entero	Rango: 25 <= x <= 30
P_CONTAINER	Cadena	Longitud: <= 8
P_RETAILPRICE	Numérico	Rango: 1500<= x <= 2000
P_COMMENT	Cadena	Longitud: <= 23

Tabla B-1. Restricciones para la tabla PART de la base de datos TPC-H.

Nombre de la tabla: SUPPLIER

Nombre de columna	Tipo de dato	Restricciones
S_SUPPKEY	Numérico	>= 5000
S_NAME	Cadena	Longitud: <= 25
S_ADDRESS	Cadena	Longitud: <= 40
S_PHONE	Cadena	Longitud: <= 15
S_ACCTBAL	Numérico	<= 4500
S_COMMENT	Cadena	Longitud: <= 101

Tabla B-2. Restricciones para la tabla SUPPLIER de la base de datos TPC-H.

Nombre de la tabla: CUSTOMER

Nombre de columna	Tipo de dato	Restricciones
C_CUSTKEY	Numérico	>= 75, 000
C_NAME	Cadena	Longitud: <= 25
C_ADDRESS	Cadena	Longitud: <= 40
C_PHONE	Cadena	Longitud: <= 15
C_ACCTBAL	Numérico	> 4495, < 9000
C_MKTSEGMENT	Cadena	Longitud: <= 9
C_COMMENT	Cadena	Longitud: <= 117

Tabla B-3. Restricciones para la tabla CUSTOMER de la base de datos TPC-H.