***Título**: Conversión de un modelo de datos para su ejecución en el cloud: del modelo relacional al no relacional*

***Volumen**: 1*
***Alumno**: Francisco Javier Navarro Alcaraz*

***Directora**: Yolanda Becerra Fontal*
***Departamento**: Arquitectura de Computadors*
***Fecha**: 25/01/2012*

DATOS DEL PROYECTO

Título del proyecto: Conversión de un modelo de datos para su ejecución en el cloud: del modelo relacional al no relacional

Nombre del estudiante: Francisco Javier Navarro Alcaraz

Titulación: Ingeniería Técnica en Informática de Sistemas

Créditos: 22,5

Directora: Yolanda Becerra Fontal

Departamento: Arquitectura de Computadors

MIEMBROS DEL TRIBUNAL *(nombre y firma)*

Presidente: Juan José Costa Prats

Vocal: M. Pilar Muñoz García

Codirector: Luis Barguñó Jané

CALIFICACIÓN

Calificación numérica:

Calificación descriptiva:

Fecha:

Agradecimientos

Quiero dar las gracias a todas las personas que me han ayudado y apoyado en acabar mis estudios y, más concretamente, este proyecto. Especialmente a mi pareja y a mi familia, mis dos pilares de la vida. Han sido 4 años muy duros y difíciles para todos y han requerido mucho esfuerzo, pero, por fin, este esfuerzo se ve recompensado.

También me gustaría mencionar a Luis Barguñó, codirector de este proyecto y creador de FUFIFU, a David Carrera, expresidente del tribunal, a Valeria Quadros, asociada al departamento de Arquitectura de Computadores, y a la directora del proyecto Yolanda Becerra ya que sin ellos realizar este proyecto no hubiera sido posible.

Índice de contenido

1. INTRODUCCIÓN

1.1. Descripción

En la actualidad, cualquier idea puede ser plasmada en una aplicación web. Esto genera un sinfín de herramientas que permiten a los usuarios de la red cubrir muchas de sus necesidades de una manera rápida y sencilla. Además, las empresas trabajan realizando mejoras en los servicios ofrecidos para aumentar el grado de satisfacción de sus usuarios.

El uso de las aplicaciones web, en la gran mayoría de casos, requiere la recolección de una serie de datos de los usuarios para su correcto funcionamiento, lo que genera la necesidad de contar con una base de datos para su buen almacenamiento y, consecuentemente, con un modelo de datos.

Las primeras aplicaciones diseñadas contaban con un número de usuarios reducido, lo que permitía que el manejo y almacenamiento de los datos de estos fuera sencillo.

La administración de estos datos requiere de un modelo de datos. Hasta ahora, el modelo más utilizado y, probablemente, el más apropiado ha sido el modelo relacional. Éste trata los datos almacenados como relaciones, es decir, como un conjunto finito de atributos que describe un objeto o una característica. Por lo tanto, según el modelo relacional una base de datos sería como una colección de relaciones.

Con la globalización y la mayor accesibilidad para el uso de los servicios e Internet, se ha generado un notable aumento de usuarios de estas aplicaciones. Esto significa un aumento considerable en las transacciones a la base de datos y en el tamaño de éstas. Por éste motivo, el modelo relacional se ve comprometido en algunas operaciones que impliquen más de una relación ya que el número de conjuntos es mucho mayor y, a la vez, cada conjunto crece aún más.

Para solventar este problema, se puede optar por escalar el sistema, es decir, ampliar las características de éste para aumentar la cantidad de trabajo que puede procesar.

Existen dos tipos de escalados: escalado vertical y escalado horizontal.

El escalado vertical es la ampliación de recursos a un único punto del sistema, tales como aumentar la memoria o cambiar la unidad de almacenamiento por una más rápida.

En cambio, el escalado horizontal es la ampliación de recursos al sistema completo, es decir, añadir nuevos puntos o nodos al sistema para distribuir el trabajo.

Hoy en día, las empresas optan por escalar horizontalmente ya que ofrece muchas ventajas:

- Distribuir la carga de trabajo entre varios puntos del sistema, aliviando la carga del mismo y no forzándolo a una posible saturación
- Es resistente a fallos de hardware como caídas de algún punto del sistema o fallos en componentes ya que se tienen alternativas
- Como los precios de los sistemas (ordenadores al fin y al cabo) disminuye y el precio de los componentes aumenta, es más económico y productivo añadir un punto o nodo más al sistema.

Dado que los diferentes gestores de bases de datos basados en el modelo relacional soportan sistemas escalables pero que su configuración, coste y complejidad no es la suficiente para que el sistema pueda trabajar con la fluidez deseada, se ha estado buscando una alternativa para éste modelo.

Esta alternativa es el modelo no relacional, más conocido por NoSQL (según la fuente, es la negación de SQL o Not only SQL). Éste modelo difiere del anterior por varios aspectos, el más significativa es que no utiliza el lenguaje SQL para hacer consultas a la base de datos

El modelo no relacional está teniendo una importante acogida ya que cada vez más empresas respaldan esta tecnología migrando sus infraestructuras a este modelo. Algunos ejemplos son Google, Amazon, Facebook, Twitter, Rackspace o CISCO.

1.2. Motivación

Dados los tiempos que corren, disponer de un sistema que sea fácilmente escalable es un punto a tener muy en cuenta, ya que el crecimiento desmesurado de la cantidad de usuarios que utilizan los servicios a través de Internet hace que disponer de un sistema con esta característica sea, prácticamente, obligatorio.

Ya que el modelo relacional no puede satisfacer esta característica, nace un nuevo modelo de datos llamado no relacional o NoSQL.

Si se busca información sobre este nuevo modelo de datos, podremos ver como muchos usuarios se resisten a utilizar el nuevo modelo, pero, a su vez, también podemos ver que grandes empresas apuestan por esta nueva tecnología.

¿Arriesgan demasiado las empresas confiando sus datos apostando por una tecnología que aún está demasiado “verde”? ¿O están esos usuarios reacios a cambiar equivocados? ¿Qué modelo es mejor?

Así pues, la motivación de este proyecto es la comparación de los dos modelos de datos, intentando responder a las preguntas antes formuladas. Para ello, convertiremos una aplicación implementada con el modelo relacional a la misma aplicación con el modelo no relacional evaluando el rendimiento de ambos.

Para la evaluación, efectuaremos operaciones de escritura, lectura y combinación de ambas sobre la base de datos midiendo los accesos a la misma y el tiempo que se emplea en servir los datos pedidos al usuario.

1.3. Planificación

1.3.1. Planificación temporal

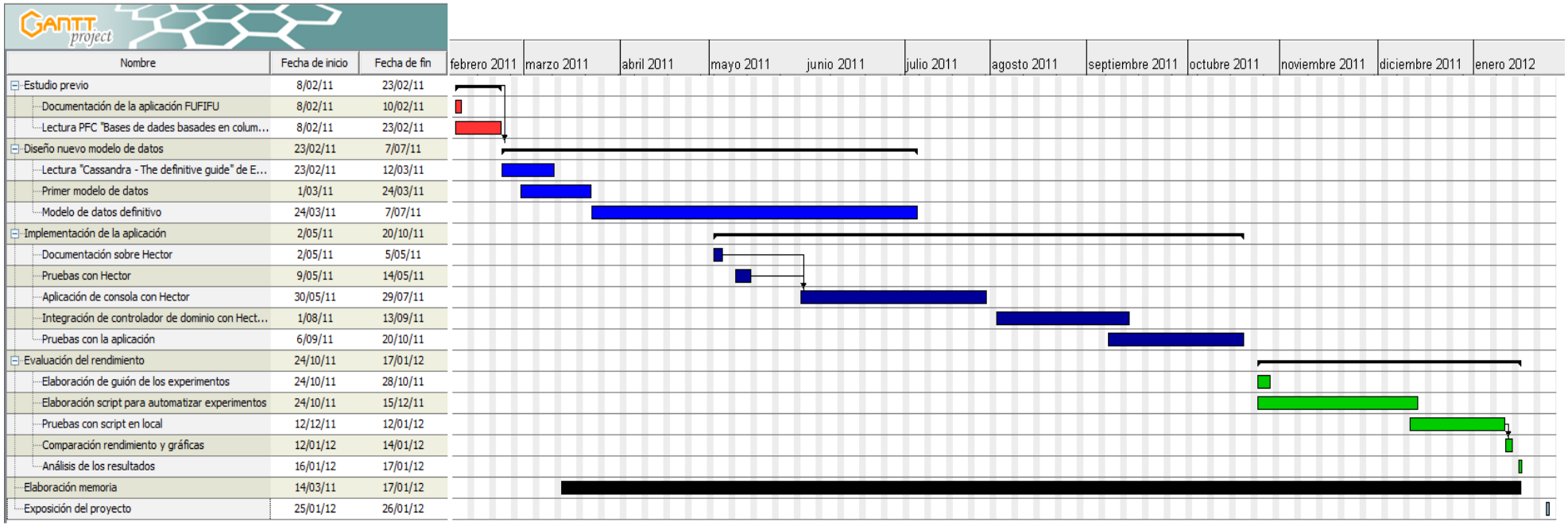
Este proyecto se empezó a elaborar el 26/01/2011 y, según se especificaba en el informe previo, estaba previsto finalizar en noviembre de ese mismo año. Esta planificación se realizó teniendo en cuenta el trabajo realizado hasta el momento y evaluando el restante.

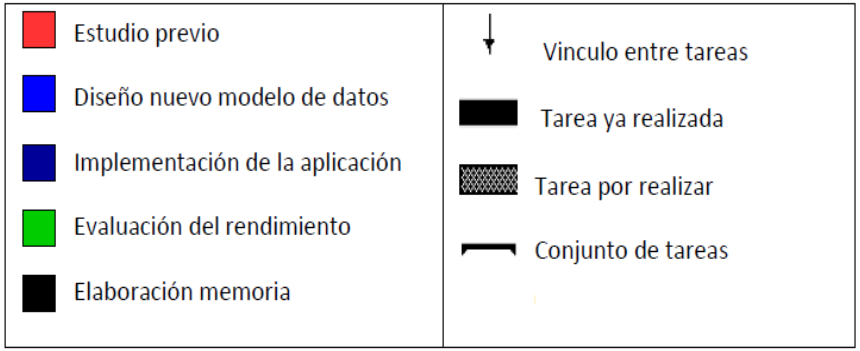
La planificación del proyecto está basada en 4 etapas: estudio previo de la nueva tecnología, diseño nuevo del modelo de datos, implementación de la aplicación y evaluación del rendimiento. Como podemos observar, todas las etapas son importantes y, sobre todo, hay que tener en cuenta que son correlativas, esto es, si no se ha acabado de diseñar el nuevo modelo de datos no es posible acabar de implementar la aplicación y, consecuentemente, no se puede evaluar el rendimiento de ésta.

Una vez diseñado el nuevo modelo de datos, se realizaron pruebas aisladas tanto con la aplicación original como con la nueva tecnología que se iba a utilizar para familiarizarme con ellos.

Una vez acabada la implementación de la aplicación, se debían realizar dos tipos de pruebas de rendimiento, detalladas en el punto 6 de éste documento, pero por causas técnicas se han visto reducidas a un solo tipo. Personalmente, me hubiera gustado realizar estas pruebas ya que nunca antes había experimentado con un sistema de éstas características.

A continuación se muestra el diagrama de Gantt definitivo que muestra la planificación real del proyecto:


GANTT project
Nombre
Fecha de inicio
Fecha de fin
febrero 2011
marzo 2011
abril 2011
mayo 2011
junio 2011
julio 2011
agosto 2011
septiembre 2011
octubre 2011
noviembre 2011
diciembre 2011
enero 2012
Estudio previo 8/02/11 23/02/11
Documentación de la aplicación FUFIFU 8/02/11 10/02/11
Lectura PFC "Bases de dades basades en colum... 8/02/11 23/02/11
Diseño nuevo modelo de datos 23/02/11 7/07/11
Lectura "Cassandra - The definitive guide" de E... 23/02/11 12/03/11
Primer modelo de datos 1/03/11 24/03/11
Modelo de datos definitivo 24/03/11 7/07/11
Implementación de la aplicación 2/05/11 20/10/11
Documentación sobre Hector 2/05/11 5/05/11
Pruebas con Hector 9/05/11 14/05/11
Aplicación de consola con Hector 30/05/11 29/07/11
Integración de controlador de dominio con Hect... 1/08/11 13/09/11
Pruebas con la aplicación 6/09/11 20/10/11
Evaluación del rendimiento 24/10/11 17/01/12
Elaboración de guión de los experimentos 24/10/11 28/10/11
Elaboración script para automatizar experimentos 24/10/11 15/12/11
Pruebas con script en local 12/12/11 12/01/12
Comparación rendimiento y gráficas 12/01/12 14/01/12
Análisis de los resultados 16/01/12 17/01/12
Elaboración memoria 14/03/11 17/01/12
Exposición del proyecto 25/01/12 26/01/12


Estudio previo
Diseño nuevo modelo de datos
Implementación de la aplicación
Evaluación del rendimiento
Elaboración memoria
Vinculo entre tareas
Tarea ya realizada
Tarea por realizar
Conjunto de tareas

Como podemos observar y al contrario de lo que he comentado antes, las etapas correlativas de diseño de modelo de datos e implementación se han solapado, pero cabe destacar que la tarea de la etapa de implementación que necesita del diseño del modelo de datos no se solapa con éste.

La planificación final varía en dos meses a la elaborada para el informe previo. Esta desviación viene dada por la comprobación del funcionamiento correcto de la aplicación y las pruebas de rendimiento.

1.3.2. Planificación económica

En la planificación económica sólo tendremos en cuenta los recursos humanos ya que como material hemos utilizado mi ordenador personal y herramientas de software libre.

La duración del proyecto ha comprendido prácticamente un año y la contabilización de las horas de trabajo ha sido de 1066 horas. El presupuesto se hace en base al precio medio de coste de hora de un analista/programador suponiendo una retribución de 25 €/hora.

Por lo tanto, el coste total del proyecto son 26.650 €

2. PROBLEMAS DEL MODELO RELACIONAL

2.1. Excesivo aumento de datos

Como he comentado en la introducción, las aplicaciones tienen cada vez más usuarios lo que implica un aumento de datos para almacenar, procesar y transferir.

Ante esta situación, los Sistemas Gestores de Bases de Datos Relacionales (SGBDR) están ofreciendo un bajo rendimiento. Esto es debido a que indexar grandes tablas supone un tiempo muy elevado de respuesta y, por lo tanto, una deficiencia en el servicio que las empresas no se pueden permitir.

Ejemplos claros de aumento excesivo de datos son empresas como Google, Facebook o Twitter que pueden llegar a generan una cantidad del orden de PetaBytes al día, cantidad elevadísima para poder gestionar por un conjunto reducido de máquinas.

Este problema puede ser solucionado mediante la escalabilidad del sistema, es decir, la capacidad de aumentar sus características sin perder la calidad del servicio.

En el caso de los SGBDR, la escalabilidad es una asignatura pendiente. Éstos son capaces de ser escalados pero la complejidad en la configuración y el coste elevado de la misma hacen que no sean una buena elección.

2.2. Alternativa

Debido a este problema, es necesaria una alternativa a éste modelo de datos ya que cada vez más empresas crecen en este mundo.

Esta alternativa nace a principios de 1999 con el término NoSQL, para agrupar a las bases de datos no relacionales.

A diferencia de la base de datos relacional, este modelo no tiene un esquema fijo, es decir, que no es necesaria la normalización de los datos teniendo así “tablas” que contengan “filas” con un número diferente de columnas. No disponen de la operación “join” por lo que se deben utilizar otros mecanismos para obtener los mismos resultados. Y, como principal ventaja sobre el modelo relacional, están diseñados para ser escalados fácilmente sin apenas configuración.

Cabe destacar que, para cumplir con todas estas características, se deben renunciar a las características ACID (Atomicity, Consistency, Isolation, Durability).

Esta elección se basa en el teorema de CAP o teorema de Brewer el cual establece que es imposible para un sistema de cómputo distribuido dar simultáneamente las características de consistencia (Consistency), disponibilidad (Availability) y tolerancia a fallos (Partition Tolerance).

La consistencia se refiere a que todos los nodos del sistema tienen que ver la misma información al mismo tiempo, es decir, que puede haber un instante de tiempo en el cual haya dos nodos que deban contener la misma información y no la contengan.

La disponibilidad hace referencia a que si uno de los nodos se cae o falla, el sistema seguiré funcionando correctamente. Es decir, que no tiene un punto único de fallo.

Y, finalmente, la tolerancia a fallos hace que el sistema siga funcionando a pesar de pérdidas arbitrarias de información o fallos parciales del sistema.

Este teorema expone que sólo se pueden satisfacer dos de las tres características al mismo tiempo.

Por lo tanto, disponemos de dos modelos de datos totalmente diferentes: mientras que el primer modelo es totalmente rígido en cuanto a las reglas de almacenamiento de datos, el segundo permite establecer las que más se adecúan a la aplicación.

2.3. Modelo no relacional

Como ya hemos comentado antes, este modelo es diferente al clásico modelo relacional.

Las diferencias, aparte de no utilizar el lenguaje SQL para consultas, son:

- La posibilidad de utilizar un esquema variable para cada "tabla" del modelo de datos, olvidándonos de los valores NULL que ocupan espacio inútilmente.
- Pérdida de las sentencias "join". Al ser ésta una operación que unía dos tablas (relaciones) a través de un atributo, en el modelo no relacional no tiene sentido tener una operación que relacione dos tablas.
- Por naturaleza, los sistemas gestores de bases de datos basados en éste modelo, son escalables horizontalmente facilitando mucho esta tarea al no tener que configurar, prácticamente, nada.

Las bases de datos basadas en este modelo suelen clasificarse según su forma de almacenar los datos. Esta clasificación está formada por 7 categorías: *Key-Value store, Document-store, Graph databases, Multivalue databases, Object databases, Tabular* y *Tuple store*.

2.3.1. Key-Value store

Este tipo de bases de datos permite diseñar un modelo de datos sin un esquema definido. Puede almacenar datos de cualquier tipo e incluso objetos utilizados en programación.

Existen varios tipos de esta categoría:

- Eventualmente consistente
  Relajan la consistencia para que, durante un tiempo pequeño, contenga información obsoleta o no actualizada. Algunos ejemplos de gestor de bases de datos de este tipo son Apache Cassandra, Dynamo o Hibari.
- Jerárquico
  El modelo de datos tiene una jerarquía, componiéndose de elementos diferenciados por nivel. GT.M o InterSystems caché son algunos ejemplos de gestores estas bases de datos.
- Key-Value en memoria
  Como su propio nombre indica, éste tipo de bases de datos operan con los datos en memoria. Algunos de ellos son memcached, OpenLink Virtuoso u Oracle Coherence.
- Key-Value almacenados en disco
  Es similar a la categoría anterior, pero en este caso se almacena los datos en disco. En esta categoría encontramos gestores tales como BigTable, CDB, MongoDB o MemcacheDB
- Key-Value ordenados
  Como describe su nombre, éste gestor permite almacenar los datos ordenados por sus keys. Algunos ejemplos Berkley DB, IBM Informix C-ISAM o MemcacheDB.

2.3.2. Document store

También llamada base de datos orientada a documento (en inglés document-oriented database). Como su propio nombre indica, está diseñada para manipular información orientada a documento.

Todas estas bases de datos tienen su propia implementación, pero tienen en común que encapsulan los documentos y codifican los datos en algún formato estándar tales como XML, YAML, JSON o BSON e incluso los binarios de los formatos PDF y los formatos de la familia de la suite ofimática Microsoft Office.

2.3.4. Graph databases

Estas bases de datos utilizan la estructura de grafos para representar y almacenar los datos contenidos.

Los nodos del grafo representarían los datos almacenados y las aristas la relación entre éstos de tal manera que, utilizando la teoría de grafos, recorramos la base de datos describiendo los atributos de los nodos y las aristas.

Deben estar totalmente normalizadas, es decir, que cada tabla tenga una sola columna y cada relación dos.

Las características son:

- Combinación de atributos multi-valor y atributos complejos
- Flexibilidad a cambios de estructura, especialmente cuando la fuente de datos es autónoma y dinámica como Internet.
- Unificación en la representación de los datos, esquemas y consultas.
- Pueden recibir o devolver grafos completos según diferentes criterios de búsqueda.

2.3.5. Multi-value

Es un tipo de base de datos multidimensional, es decir, tiene una estructura para organizar los datos en diferentes dimensiones. Son parecidas a las bases de datos relacionales, diferenciándose en que almacenan más de un valor en cada celda de una tabla.

A diferencia de los otros tipos de base de datos NoSQL, éste soporta el lenguaje de bases de datos relacionales SQL.

2.3.6. Object database

En estas bases de datos, la información se representa como un objeto como los presentes en la programación orientada a objetos.

Cuando se integra las características de una base de datos con las de un lenguaje de programación orientado a objetos, el resultado es un sistema gestor de bases de datos orientada a objetos (SGBDOB). Un SGBDOB representa los datos de la base de datos como si se tratase de un objeto de un lenguaje de programación.

Estas bases de datos están pensadas para trabajar con lenguajes de programación orientados a objetos como Java, C#, Visual Basic .NET y C++.

Dispone de tres tipos de características: obligatorias o fijas, opcionales y abiertas.

Las características obligatorias son las habituales de los lenguajes de programación orientados a objeto y de las bases de datos, tales como encapsulación de datos, tipos y clases de objetos, herencia, identidad única de los objetos, persistencia de las operaciones, integridad de los cálculos o concurrencia. Además, permite la ampliación

del tipo de datos por parte del usuario, overriding y overloading[1], gestión de almacenamiento secundario y recuperación.

Las características opcionales se pueden añadir para que la base de datos sea más eficiente y lograr una mayor funcionalidad. Éstas son herencia múltiple, comprobación de tipos e inferencia, distribución, operaciones de diseño y versiones.

Las características abiertas son aquellas que tratan los temas de programación y representación. Junto con las otras dos características anteriores, ofrece muchas opciones para mejorar el modelo y construir un buen sistema.

[1] Overriding y overloading, en este caso, es la posibilidad de utilizar un mismo nombre para implementar métodos u operaciones que representen los mismos datos pero en diferentes formas.

3. CASSANDRA

3.1. Historia

Cassandra fue desarrollado por Avinash Lakshman, uno de los creadores de Dynamo de Amazon, y Prashant Malik, ingeniero de Facebook, para potenciar la herramienta de Inbox Search de Facebook.

Facebook publicó esta herramienta como open-source en julio de 2008 en la web para desarrolladores Google Code y en marzo de 2009 se convierte en un proyecto Apache Incubator, siendo finalmente proyecto top-level en febrero de 2010. Facebook abandona el proyecto Cassandra y deja de utilizarlo a finales de 2010 ya que su nueva plataforma de mensajería se desarrolla en HBase.

A partir de entonces, Apache Software Foundation es la encargada de seguir desarrollando éste software juntamente con la gran comunidad de desarrolladores que tiene a sus espaldas.

3.2. ¿Qué es Cassandra?

Cassandra es un sistema gestor de bases de datos distribuidas de código abierto y está diseñado para manipular grandes cantidades de datos a través de varios servidores proporcionando un servicio de alta disponibilidad sin puntos únicos de fallo.

Es del tipo Key-Value store, de la categoría eventualmente consistente. Es configurable por el usuario para poder modificar la consistencia y la disponibilidad de los datos y, por lo tanto, tener un rendimiento del sistema mayor.

3.3. Modelo de datos

Antes de explicar cómo es el modelo de datos de Cassandra, hay que tener en cuenta las siguientes definiciones:

- Clúster: es el conjunto de máquinas, también llamadas nodos, que forman la unidad lógica de Cassandra.
- Se describirán los elementos del modelo de datos por niveles, siendo el nivel más alto el *Keyspace* (equivalente a una base de datos en el modelo relacional) y el nivel más bajo la *Column* (equivalente a una columna en el modelo relacional).

A continuación se describe el modelo de datos de Cassandra y, para entenderlo mejor, se indicará la equivalencia en el modelo relacional.

Column

Es el nivel más bajo del modelo de datos de Cassandra. Está compuesta por 3 elementos: *column name*, *value* y *timestamp*.

Column name es el identificador con el que podremos acceder a ella para obtener o modificar el valor que contiene. Es único y no puede haber dos iguales en el mismo conjunto de columnas.

Value es el dato de una columna. Es el único elemento modificable por el usuario. Se puede definir la validación de éste valor estableciendo así el tipo de dato que contiene. Hay varios tipos de datos que acepta la validación entre los cuales se encuentran la codificación UTF-8, el tipo Long o el tipo Byte. Por defecto se establece el tipo Byte.

Timestamp nos indica cuándo se modificó por última vez esa columna. Éste elemento no es modificable por el usuario ya que se genera automáticamente al cambiar el campo *value*. Éste campo es único para las *columns.*

Se representa en JavaScript Object Notation (JSON) de la siguiente manera:

```
"Column 1":{
    "name": "Nombre de la columna",
    "value": "Valor de la columna",
    "timestamp": 123456789
}
```

Su equivalente en el modelo relacional es una columna de una fila de una tabla.

Row

Es el siguiente nivel en el modelo de datos. Es un conjunto de *columns* identificado por una *key* o *clave* para acceder a él y es única en el conjunto de *rows* en el que se encuentra. Se puede establecer el orden en el que se almacenan las *columns* que contiene por el campo *column name* utilizando un comparador que, al igual que en el caso del atributo *value* de las *columns*, acepta varios tipos de datos.

A diferencia del modelo relacional, dos *rows* de un mismo conjunto pueden tener diferente número de columnas y no es necesario tener columnas vacías que ocupen espacio en disco.

La representación de una *row* en JSON sería de la siguiente manera:

```
"Row 1":{
    "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"},
    "Column 2":{"name":"Nombre Columna 2", "value":"Valor Columna 2"}
}
```

Su equivalente en el modelo relacional es una fila de una tabla.

Column Family

El siguiente nivel en el modelo de datos de Cassandra es la *Column Family*. Ésta contiene una colección de *rows* que podrán ser accedidas por su *row key* y está identificada por un nombre, el cual será único.

Al igual que las *rows*, las *Column Families* se pueden configurar para que almacenen las *rows* que contienen en un determinado orden, siguiendo exactamente el mismo patrón que en el nivel anterior.

La representación de una *Column Family* en notación JSON sería la siguiente:

```
"Column Family 1":{
    "Row 1":{
        "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"},
        "Column 2":{"name":"Nombre Columna 2", "value":"Valor Columna 2"}
    },
    "Row 2":{
        "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"},
        "Column 2":{"name":"Nombre Columna 2", "value":"Valor Columna 2"},
        "Column 3":{"name":"Nombre Columna 3", "value":"Valor Columna 3"}
    },
    "Row 3":{
        "Column 3":{"name":"Nombre Columna 3", "value":"Valor Columna 3"}
    }
}
```

Como podemos observar, tenemos 3 *rows* con diferentes columnas cada una. Esto en el modelo relacional no se contempla ya que, obligatoriamente, todas las filas deben tener las mismas columnas.

Su equivalente en el modelo relacional es la tabla.

Existe un tipo de *Column Family* denominado *Super Column Family*. Añaden un nivel más a las *Column Families*. Se podría decir que contienen *Column Families*, pero en éste caso se denominan *Super Columns* teniendo el mismo comportamiento que las *Column Families*.

Las *Super Column Families* tienen dos comparadores para establecer el orden en el que se almacenan las *rows* y en el que se almacenan las *columns* que éstas contienen.

Su representación en JSON es la siguiente:

```
"Super Column Family 1":{
    "Super Column 1":{
        "Row 1":{
            "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"},
            "Column 2":{"name":"Nombre Columna 2", "value":"Valor Columna 2"}
        },
        "Row 2":{
            "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"},
            "Column 2":{"name":"Nombre Columna 2", "value":"Valor Columna 2"},
            "Column 3":{"name":"Nombre Columna 3", "value":"Valor Columna 3"}
        },
        "Row 3":{
            "Column 3":{"name":"Nombre Columna 3", "value":"Valor Columna 3"}
        }
    },
    "Super Column 2":{
        "Row 4":{
            "Column 4":{"name":"Nombre Columna 4", "value":"Valor Columna 4"},
            "Column 5":{"name":"Nombre Columna 5", "value":"Valor Columna 5"}
        },
        "Row 5":{
            "Column 5":{"name":"Nombre Columna 5", "value":"Valor Columna 5"}
        }
    }
}
```

La *Super Column Family* no tiene equivalente en el modelo relacional.

Keyspace

El keyspace es el nivel más alto del modelo de datos. En él se definen todas las *Column* y *Super Column Families* que se necesiten.

Representación en JSON de un *keyspace*:

```
"Keyspace 1":{
    "Super Column Family 1":{
        "Super Column 1":{
            "Row 1":{
                "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"},
                "Column 2":{"name":"Nombre Columna 2", "value":"Valor Columna 2"}
            },
            "Row 2":{
                "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"},
                "Column 2":{"name":"Nombre Columna 2", "value":"Valor Columna 2"}
            }
        },
        "Super Column 2":{
            "Row 4":{
                "Column 4":{"name":"Nombre Columna 4", "value":"Valor Columna 4"},
                "Column 5":{"name":"Nombre Columna 5", "value":"Valor Columna 5"}
            },
            "Row 5":{
            "Column 5":{"name":"Nombre Columna 5", "value":"Valor Columna 5"}
        }
    },
    "Super Column Family 2":{
        "Super Column 1":{
            "Row 1":{
                "Column 1":{"name":"Nombre Columna 1", "value":"Valor Columna 1"}
            }
        },
        "Super Column 2":{
            "Row 5":{
                "Column 5":{"name":"Nombre Columna 5", "value":"Valor Columna 5"}
            }
        }
    }
}
```

Al definir el *Keyspace*, se pueden configurar los siguientes parámetros:

- *Partitioner*: éste parámetro especifica cómo se almacenarán las *rows* mediante su key, lo cual nos permitirá decidir cómo se distribuirán las *rows* en los nodos. Éste *partitioner* sólo afecta a la distribución de las *rows* y no a la ordenación de las *columns* contenidas en ellas.
- *Replication factor*: éste parámetro establece el número de nodos que actuarán como copias de un conjunto de datos, normalmente *rows*, es decir, cuántas veces estarán repetidos los datos en el clúster.
- *Placement strategy*: éste parámetro establece el modo en el que se replicarán los datos en el clúster. Tiene varias opciones para escoger. Para definir éstas opciones, imaginémonos que disponemos de 2 *data center* y que cada uno de ellos dispone de 2 *racks*:
    - *SimpleStrategy*: sólo se replican los datos en un *data center* y sin tener en cuenta el *rack* donde se coloca. Esto significa que si se modifican datos en el nodo 1 del *rack* 1 también se modificarán en el nodo 1 del *rack* 1. También hay que tener en cuenta el *replication factor*. Si, por ejemplo, fuera 2, se copiaría al nodo cuyo *token* es más parecido al nodo original del *rack* y, por consecuencia, al mismo nodo del otro *rack*. Éste método

es muy rápido siempre y cuando el nodo con *token* más parecido (se determina por el *partitioner*) esté en el mismo *rack*. Es el valor por defecto cuando se configura el *keyspace*.

- *OldNetworkTopologyStrategy*: se puede especificar para cada *data center* cuántas réplicas habrán en función de cada *keyspace*. Básicamente se utiliza para que las réplicas se hagan en diferentes *data centers*. Se usa normalmente cuando el *replication factor* es N+1, siendo N el número de *data centers*, y tenemos nodos del mismo clúster de Cassandra repartidos en los N *data centers*. Es necesario un *snitch* que determinará la posición relativa de los otros nodos.
- *NetworkTopologyStrategy*: es muy parecido a *OldNetworkTopologyStrategy* salvo que permite especificar cómo se repartirán las réplicas en cada *data center*.

Su equivalente en el modelo relacional es la base de datos.

3.4. Características de Cassandra

Como ya he dicho antes, Cassandra es eventualmente consistente y esto es debido a que podemos configurar el número de copias que tendremos de los datos. Esto quiere decir que, dependiendo de cómo configuremos Cassandra, la base de datos podrá contener datos replicados que durante un tiempo no sean iguales, por lo que para una misma consulta se pueden obtener diferentes resultados.

3.4.1. Nivel de consistencia

El nivel de consistencia se diferencia entre las escrituras y las lecturas. Las nomenclaturas son idénticas, pero varía el significado según la operación.

En las escrituras tenemos los siguientes niveles de consistencia:

ANY
Éste nivel de consistencia asegura que la escritura se realizará en, al menos, un nodo antes de responder al cliente con el resultado de la operación.

ONE
Éste nivel de consistencia asegura que la escritura se realizará en, al menos, el log de commit y la tabla de memoria de una réplica antes de responder al cliente con el resultado de la operación.

TWO
Éste nivel de consistencia asegura que la escritura se realizará en, al menos, el log de commit y la tabla de memoria de dos réplicas antes de responder al cliente con el resultado de la operación.

THREE
Éste nivel de consistencia asegura que la escritura se realizará en, al menos, el log de commit y la tabla de memoria de tres réplicas antes de responder al cliente con el resultado de la operación.

QUORUM
Éste nivel de consistencia asegura que la escritura se realizará en el log de commit y la tabla de memoria de $\frac{N}{2}+1$ réplicas antes de responder al cliente con el resultado de la operación, siendo N el número total de nodos que forman el clúster.

LOCAL QUORUM
Éste nivel de consistencia asegura que la escritura se realizará en el log de commit y la tabla de memoria de $\frac{R}{2}+1$ nodos en el clúster del datacenter en el que se están modificando los datos antes de responder al cliente con el resultado de la operación, siendo R el factor de replicación.
Éste nivel de consistencia tiene sentido si la base de datos está distribuida en varios datacenters.
Es necesario configurar el parámetro *Placement strategy* con el valor *NetworkTopologyStrategy*.

EACH QUORUM
Éste nivel de consistencia asegura que la escritura se realizará en el log de commit y la tabla de memoria de $\frac{R}{2}+1$ nodos en el clúster de cada datacenter antes de responder al cliente con el resultado de la operación, siendo R el factor de replicación.
Éste nivel de consistencia tiene sentido si la base de datos está distribuida en varios datacenters.
Es necesario configurar el parámetro *Placement strategy* con el valor *NetworkTopologyStrategy*.

ALL
Éste nivel de consistencia asegura que la escritura se realizará en el log de commit y la tabla de memoria de las N replicas antes de responder al cliente con el resultado de la operación. Hay que tener en cuenta que si alguna de estas réplicas falla, la operación también fallaría.

En las lecturas tenemos los siguientes niveles de consistencia:

ONE
Éste nivel de consistencia devolverá el resultado obtenido de la primera réplica que responda.
Para éste nivel debemos saber que en segundo plano se estará ejecutando una comprobación de consistencia que su función es buscar la réplica más reciente de los datos, por si hubiera algún dato obsoleto y actualizarlo con el más reciente. Con esta comprobación, en operaciones de lectura posteriores se obtendrá el resultado correcto incluso si la lectura inicial obtuvo un valor antiguo y, por lo tanto, no válido. A esta operación se le llama *ReadRepair*.

TWO
Éste nivel de consistencia consultará en dos réplicas y devolverá el dato más reciente.
De nuevo, la comprobación de la consistencia se realizará en segundo plano.

THREE
Éste nivel de consistencia consultará en tres réplicas y devolverá el dato más reciente.
De nuevo, la comprobación de la consistencia se realizará en segundo plano.

QUORUM
Éste nivel de consistencia consultará todas las réplicas y devolverá el dato más reciente una vez se hayan consultado, como mínimo, $\frac{N}{2}+1$ réplicas.
De nuevo, la comprobación de la consistencia se realizará en segundo plano.

LOCAL QUORUM
Éste nivel de consistencia devolverá el resultado más reciente una vez se hayan comprobado la mayoría de las réplicas del *datacenter* en el que se están consultando los datos.
Éste nivel de consistencia tiene sentido si la base de datos está distribuida en varios datacenters.

EACH QUORUM
Éste nivel de consistencia devolverá el resultado más reciente una vez se hayan comprobado la mayoría de las réplicas de cada *datacenter* antes de responder al cliente.
Éste nivel de consistencia tiene sentido si la base de datos está distribuida en varios datacenters.

ALL
Éste nivel de consistencia devolverá el resultado más reciente una vez se hayan comprobado todas las réplicas antes de devolver el resultado al usuario.
Hay que tener en cuenta que si alguna de estas réplicas falla, la operación también fallaría.

3.4.2. Secondary Index

Una de las características que se incluyeron a partir de la versión 0.7 es el Secondary Index.

Este tipo de índices permite realizar consultas a Cassandra filtrando los resultados por un value de una column en concreto utilizando predicados de igualdad tal y como se hace en los gestores de bases de datos relacionales (where id=10, por ejemplo). Se les llama secundario para diferenciarlos de la row key que tienen las Column Families que, normalmente, son tratados como índices.

De momento, sólo soporta el tipo de índice KEYS que es similar a un índice de hash. Actualmente, están desarrollando el tipo de índice BITMAP que es una estructura muy eficiente para tratar con datos inmutables, es decir, que una vez creados no se modificarán.

Los Secondary Index son una buena elección cuando varias rows contienen el valor indexado. En cambio, para rows que contienen valores únicos es más eficiente realizar este indexado mediante otra Column Family.

Una de las ventajas de utilizar el Secondary Index de Cassandra, es que al añadir uno a una Column Family ya existente y poblada, automáticamente indexa los datos existentes en background.

Se construyen automáticamente en background y, además, sin bloquear las operaciones de lectura ni escritura.

4. CLIENTES PARA LA COMUNICACIÓN CON CASSANDRA

El desarrollo de aplicaciones es siempre en lenguajes de programación de alto nivel. Estas aplicaciones suelen necesitar una base de datos para almacenar información y, para hacer más fácil la comunicación entre la aplicación y la base de datos, se implementan clientes o librerías para dicha comunicación.

Gracias a la gran comunidad de programadores que se ha interesado en implementar estos clientes, disponemos de varias opciones para los lenguajes de programación más utilizados.

4.1. Diferentes vías de comunicación

A continuación, veremos las diferentes formas de comunicarnos con la base de datos de Cassandra.

Hemos separado los clientes en dos bloques: el primero es el cliente que se proporciona con Cassandra y el segundo son los clientes más utilizados en lenguajes de programación de alto nivel.

Esta información es para los clientes compatibles para la versión 0.7 de Cassandra. En la actualidad, se ha ampliado la lista y se han mejorado características de los clientes.

4.1.1. Cliente de Cassandra

Cassandra se distribuye con un cliente básico para la comunicación con la base de datos. Éste cliente permite realizar las operaciones de inserción, modificación y eliminación de los elementos del modelo de datos de Cassandra.

Está construido a nivel de *Thrift* y, por éste motivo, carece de las características más comunes en los clientes para bases de datos tales como el conection pooling o el monitorizaje.

Thrift es un Interface Description Language (IDL) que es utilizado para definir y crear servicios para numerosos lenguajes de programación. Se utiliza como framework para realizar Remote Procedure Call (RPC), es decir, como un proceso que se comunica con otro proceso (en éste caso el de Cassandra), normalmente en otra máquina, a través de un sistema de bajo nivel de mensajería para que ejecute código en la otra máquina. Fue creado por Facebook para el desarrollo de servicios escalables entre diferentes lenguajes de programación.

4.1.2. Clientes en lenguaje de alto nivel para Cassandra

A continuación se describen las características más relevantes de los diferentes clientes en lenguaje de alto nivel para la comunicación con Cassandra.

4.1.2.1. Java

En el lenguaje Java tenemos tres opciones principales:

1. **Hector**

   Hector fue creado para añadir las características de las que no dispone el cliente suministrado con Cassandra.

   Su autor es Ran Tavory y es una API con licencia MIT y código open-source.

   Sus características principales son:

   · Interfaz orientada a objetos para Cassandra
   · Failover en el lado del cliente
   · Connection pool para mejorar el rendimiento y la escalabilidad de operaciones a la base de datos
   · Monitorización y gestión a través de JMX
   · Balanceo de carga configurable y expandible con 3 algoritmos (round robin, least active y un detector de tiempo de respuesta phi-accrual)
   · Encapsulamiento total de la API Thrift y structs
   · Reintentos automáticos para hosts caídos
   · Detección automática de nuevos hosts en el clúster
   · Suspensión de hosts por un tiempo después de varios intentos fallidos de conexión
   · Capa simple de Object-Relational Mapping (convertir los objetos de las clases de programación en objetos de la base de datos)
   · Un acceso seguro para tratar el modelo de datos de Cassandra

2. **Kundera**

   La idea principal de esta API es hacer más amigable el funcionamiento de las bases de datos NoSQL (no relacionales).

   Kundera es una JPA compatible con el almacenamiento de datos a través del mapeo de objetos para las bases de datos NoSQL, es decir, transforma los objetos de las clases de programación en los datos para ser introducidos en la base de datos.

   Actualmente soporta Cassandra, HBase y MongoDB conjuntamente con las bases de datos relacionales.

   Kundera aprovecha las librerías ya existentes y construye una API abstracta para facilitar a los desarrolladores el no utilizar código repetitivo e innecesario, creando así código más ordenado y limpio reduciendo la complejidad y mejorando la calidad.

   Kundera soporta la persistencia a través del almacenamiento de datos. Esto significa que podemos guardar u obtener objetos relacionados en diferentes bases de datos usando una única función.

   Todavía no gestiona las transacciones. Por lo tanto, es responsabilidad del programador asegurar la atomicidad de las operaciones de su aplicación.

3. **Pelops**

   Esta API se creó para mejorar la calidad de Cassandra a través de un proyecto complejo que hacía un gran uso de las bases de datos.
   Los principales objetivos de Pelops son:

· Hacer más fácil el entendimiento de la API de Cassandra
· Aislar completamente la parte de bajo nivel de la parte de alto nivel como, por ejemplo, el connection pooling del código de procesamiento de datos.
· Eliminar el "dressing code", por lo que la semántica del procesamiento de datos queda clara y evidente.
· Acelerar el desarrollo a través de IntelliSense, sobrecargando las funciones y potentes métodos de alto nivel.

Disponemos de dos opciones más en lenguaje Java: Datanucleus JDO y Cassandrelle. Datanucleus JDO es un plugin que ejerce de JPA para la base de datos de Cassandra. Se ha sustituido por Hector JPA ya que los requerimientos para la implementación de una JPA eran significativamente superiores a los que éste plugin ofrecía.
Cassandrelle (Demoiselle Cassandra) es otro plugin para la JPA de las bases de datos de Cassandra para Demoiselle. Demoiselle es un Framework formado por seis proyectos que está pensado para desarrollar aplicaciones web con los siguientes objetivos: estandarizar el desarrollo de las aplicaciones, generar código reutilizable, ser abierto y compartido, permitir el desarrollo con diferentes colaboraciones (repartir el trabajo) y permitir la integración de diferentes tecnologías.

4.1.2.2. Python

En el lenguaje Python tenemos tres opciones:

1. **Pycassa**
   Pycassa es una librería para el lenguaje de programación Python que facilita la integración de las bases de datos de Cassandra con éste lenguaje.
   Sus características principales son:
   · Failover y reintentos de operaciones automático.
   · Connection pooling
   · Soporte para multithread
   · Batch interface, es decir, una vez especificado el trabajo a realizar se ejecuta y el usuario no tiene que interactuar para su funcionamiento.
   · Puede mapear clases de objetos a *Column Families* de Cassandra.
   Es la mejor opción para éste lenguaje ya que tiene las características necesarias para trabajar cómodamente con Cassandra.

2. **Telephus**
   Telephus es una API de bajo nivel y con connection pooled para Cassandra desarrollada en Twisted.
   Twisted es un Framework de red para programación dirigida por eventos, lo cual significa que se pueden escribir pequeños *callbacks* que serán llamados por el Framework.
   Al no estar muy utilizado, no se dispone de mucha información.

3. **Django Backend para Cassandra**
   Éste cliente no es la mejor opción en Python ya que la última versión publicada es la 0.2. Aún está en desarrollo y tiene muchos fallos, algunos conocidos pero no corregidos. Por ello no es recomendable utilizar esta opción en un entorno de producción.

Actualmente, funcionan correctamente las siguientes operaciones:
· Operaciones básicas como la creación de modelos, consultas, conteo, actualización, guardado, eliminado y ordenado.
· Consultas eficientes para las coincidencias exactas para la *primary key* (sería la *key* para las *column families* y la *super key* para las *super column families*). También funcionan correctamente las *range queries*, siempre y cuando esté configurado el clúster con el particionador *OrderPreservingPartitioner*.
· Consultas ineficientes para las demás operaciones que no se puedan realizar eficientemente en Cassandra.
· Creación de *keyspace* y *column families* a través de *syncdb*.
· Interfaz de usuario administrativa de Django.
· Operaciones con filtrado de datos como *startswith*, *regex*, etc.
· Operaciones complejas con Q nodos.
· Soporte básico para el *secondary index*. Si el atributo *db_index* de un campo está configurado, en *backend* se configura la *column family* para indexar por ese campo. En la versión 0.7.0 de Cassandra sólo está soportado coincidencias exactas para las búsquedas por *secondary index*. Esta operación mejorará con versiones posteriores de Cassandra, por lo que ésta característica podríamos decir que es temporal.
Las operaciones que no funcionan o que no se han comprobado actualmente son:
· Todos los tipos de datos para los diferentes campos (*column name*, *value*, *key*...) no se han comprobado. Puede haber problemas a la hora de convertir datos para guardar u obtenerlos de Cassandra.
· Operaciones de *join*.
· Operaciones fragmentadas. En la última versión se intenta realizar toda la operación en una vez.
· ListModel / ListField de djangotoolbox. No se ha investigado cómo funciona ni si Cassandra lo soporta. Se está trabajando en ello para que funcione correctamente.
· No es posible configurar el nivel de consistencia en las lecturas ni en las escrituras a la hora de hacer una operación.
· La autenticación de Cassandra. No está comprobado, pero probablemente funcione correctamente.

Las funcionalidades más importantes que están pendientes de revisión son:
· No es posible acceder a la interfaz del usuario administrador en la autenticación de Django.
· Problemas al utilizar *strings* codificados en UNICODE. Hay dos posibles causas de éste problema: el backend de Django o la comunicación entre Python y el *thrift* de Cassandra. El autor ha trabajado en ello, pero aún falla cuando se utilizan caracteres non-ASCII (los caracteres que se codifican con los números 128 a 255).
· Existen varias unidades de test que fallan cuando actúa el middleware. Esto no ocurre con otros backends nonrel, por lo que es un fallo o limitación del backend de Cassandra.
· Si se activa la autenticación y la sesión en middleware, fallará un grupo de la unidad de test si se ejecutan todas las unidades.

4.1.2.3. Ruby

En Ruby sólo disponemos de una opción.

1. **Fauna**

   No se dispone de mucha información y en la web de http://fauna.org/ sólo se muestran las siguientes características:
   - Encapsulación de la API Thrift
   - Compatible con los tipos de datos UUID y Long para la generación de GUID
   - Compatibilidad con Ruby 1.9
   - Automáticamente añade nodos sin necesidad de configuración

4.1.2.4. PHP

En PHP tenemos dos opciones:

1. **PHPcassa**

   PHPcassa es una librería para Cassandra.
   Aunque es muy utilizado, la información de ésta librería es muy escasa. En DataStax describen las características más relevantes de ésta librería:
   - Connection pooling para un mayor rendimiento
   - Métodos para contabilizar *rows*
   - Soporte para *Secondary Indexes* de Cassandra

   Por lo que se puede observar en los códigos de ejemplos, puede realizar, a parte de las operaciones básicas de insertado/consulta/eliminación, consultas con rangos de *keys* y establecer el tipo de comparador para *columns* y *rows*.

2. **SimpleCassie**

   SimpleCassie es un paquete totalmente independiente de la API de Thrift ya que está integrado en éste.
   En su propia Wiki se puede consultar un RoadMap[2] de desarrollo:
   - Conexión con balanceo de carga
   - Próximamente soportará consultas con rangos de *keys*
   - Eliminar varias *rows*/*keys* a la vez
   - Incrementar y decrementar varias *rows/keys* a la vez
   - Truncar datos
   - Descripción de todos los métodos o funciones
   - Batch mutate: modificaciones de la base de datos por lotes

4.1.2.5. .NET

En .NET tenemos dos opciones:

1. **Aquiles**

   Aquiles es un cliente desarrollado en .NET en su versión 3.5 o superior para Cassandra compatible con la versión 0.6 o superior.
   Básicamente, éste cliente nos ofrece las siguientes características:
   - Una interfaz en .NET para operar con Cassandra como si se hiciese con SQL
   - Connection Pool con calentamiento y control de tamaño

[2] RoadMap se podría traducir como hoja de ruta y se suele utilizar en el desarrollo de software para marcar objetivos a cumplir.

- Un gestor para controlar la entrada de nuevos nodos a un clúster llamado Endpoint Manager. Automáticamente distribuye las conexiones entre todos los nodos que constituyan el clúster y, además, comprueba la conectividad con los nodos por si hubiera alguna caída
- Una configuración simple y fácil para el usuario para configurar todos los clústers y con soporte para frameworks con depency-injection[3]
- Posibilidad de controlar más de un clúster en aplicaciones
- Posibilidad de escoger qué Connection Pool se va a utilizar y sus parámetros internos
- Posibilidad de escoger qué EndPoint Manager se va a utilizar
- Posibilidad de escoger qué clase de Transport se va a utilizar. En el caso de utilizar el transporte *TBufferedTransport*, se puede configurar incluso la longitud del buffer
- Posibilidad de añadir, modificar o eliminar *Keyspaces* y *ColumnFamilies*
- Byte Encoder para no tener que crear uno propio para tipos de datos como Long, UTF8, ASCII, GUID, UUID, etc.
- Monitorizaje de las conexiones a Cassandra mediante un monitor (nativo en Windows) implementado en PerformanceCounterHelperfwk

Es la mejor opción para éste lenguaje de programación ya que dispone de una gran comunidad de programadores.

2. **FluentCassandra**
   FluentCassandra es una librería desarrollada en .NET para acceder a Cassandra. Implementa todos los comandos para la comunicación con Cassandra y, además, soporta dynamic keyword de .NET 4.0 así como LINQ para consultas a la base de datos.

   La meta de ésta librería es mantener el interfaz sincronizado con la última versión de Cassandra y hacer lo más fácil posible a los programadores de .NET el empezar a utilizar Cassandra como base de datos.

4.2. Opción elegida

Para realizar mi proyecto, era necesario uno de estos clientes para la comunicación con la base de datos ya que realizar las operaciones con el cliente básico que se distribuye con Cassandra sería un trabajo muy complicado y laborioso.

Para decidir qué cliente escoger he tenido en cuenta los siguientes requisitos:

1. Fácil integración con la aplicación FUFIFU.
   Ya que FUFIFU está desarrollada íntegramente en Java, para una mejor integración una buena opción sería un cliente en el mismo lenguaje. Así, y gracias a que FUFIFU está organizada por paquetes, únicamente deberemos cambiar el paquete correspondiente a la comunicación con la base de datos.

[3] Dependency-injection es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en vez de ser la propia clase quien cree el objeto.

2. Soporte para problemas.
   Es muy importante que el cliente disponga de un soporte o ayuda para posibles problemas o dudas que puedan surgir. Este soporte podrían ser foros, comunidades de desarrolladores o, incluso, empresas que lo ofrezcan.

3. Características propias del cliente.
   Al utilizar un cliente para la conexión a base de datos, se busca que éste sea fácil de utilizar y que sus características se adecuen a los requerimientos de cada uno. Por eso,

Una vez se han examinado los clientes y comprobado los tres puntos anteriores, la opción elegida es Hector ya que cumple con todos los requisitos antes descritos.

5. FUFISSANDRA

5.1. Descripción

5.1.1. ¿Qué es FUFISSANDRA?

FUFISSANDRA es la adaptación de la aplicación web FUFIFU al modelo no relacional de las bases de datos.

Esta adaptación requiere cambios estructurales en la aplicación ya que, con el nuevo modelo de datos, algunas operaciones pueden ser demasiado costosas de implementar. Más adelante se detallarán estos cambios.

Para llevarla a cabo, se han de conocer las consultas que se realizan en la aplicación original. Esto conllevo un estudio exhaustivo de la misma, extrayendo toda la información posible tanto de las consultas como del código escrito ya que los métodos o servlets realizan llamadas a funciones en las que interviene la base de datos.

Tal y como está diseñado FUFIFU, dispone de una clase encargada de gestionar la comunicación entre la interficie y la base de datos. Esta clase se llama controlador de dominio. En la adaptación, se han mantenido los nombres de las funciones de ésta para que el código de las demás no se vea afectado y, así, tener la máxima similitud entre ambas aplicaciones.

Además, ha sido necesario crear clases que en FUFIFU no estaban como Intercambio o EstoyBuscando y también suprimir como NodoTrueque y PalabraOferta. Más adelante, se mostrará un diagrama de clases y una imagen de la base de datos de ambos modelos.

5.1.2. ¿Qué es FUFIFU?

FUFIFU es el proyecto final de carrera de Luis Barguñó Jané, exalumno de la Facultad de Informática de Barcelona (FIB).

Según describe su autor, FUFIFU es una red social de intercambio de bienes y servicios sin dinero, utilizando como moneda de intercambio el tiempo.

El principio es muy simple: todos tenemos habilidades y podemos ofrecer nuestro tiempo para compartirlas con los demás.

5.1.3. Funcionamiento de la aplicación

El funcionamiento es parecido al de un Banco de Tiempo. Un usuario que solicite un servicio a otro usuario, le "pagará" con la moneda propia de FUFIFU, llamada T (de Tiempo). Cada usuario podrá usar sus T's para acceder a servicios ofrecidos por el resto de usuarios. De esta manera, la moneda de cambio será el Tiempo y no el dinero.

A diferencia de un Banco de Tiempo tradicional, FUFIFU incorpora también intercambio de bienes, de manera que quien quiera podrá ofrecer bienes a cambio de T. Igualmente, cualquier usuario podrá utilizar sus T's para acceder a bienes ofrecidos por el resto de usuarios.

5.2. Cambios estructurales

Estos cambios se han llevado a cabo debido a que Cassandra está optimizado para un excelente rendimiento en escritura de datos y no para la lectura. Por lo tanto, debemos reducir el número de consultas a la base de datos.

A continuación, se detallan los cambios realizados para la aplicación FUFISSANDRA.

5.2.1. Orden de las ofertas

FUFIFU permite ordenar las ofertas de cada usuario por fecha de creación, valor, intercambios o puntuación. Esto es gracias a que el modelo relacional permite filtrar las búsquedas por cada columna de una tabla, siendo en éste caso las columnas mencionadas antes.

Por simplicidad, he decidido quitar el orden por valor, intercambios y puntuación ya que implicaría una consulta extra a la base de datos a la hora de mostrarle al usuario las ofertas que posee.

Para implementarlo en el modelo no relacional, harían falta tres *Super Column Families*, una para cada criterio de ordenación y tener indexadas los identificadores de oferta por su valor, sus intercambios y sus puntuaciones, respectivamente.

Para el orden por fecha he utilizado el identificador ya que cuanto mayor sea éste, más actual será la oferta.

5.2.2. Notificación de mensajes

FUFIFU dispone de un sistema de notificación para los mensajes que aún no han sido leídos por el usuario.

El funcionamiento es sencillo: cada vez que se escriba un mensaje nuevo en una conversación, se modificará la tabla que contiene los mensajes por leer. Una vez el usuario esté en FUFIFU, se consultará dicha tabla filtrando por el usuario y se contabilizarán todas las filas que contiene esa consulta.

En FUFISSANDRA se ha modificado para que contabilice las conversaciones con mensajes nuevos que no haya leído el usuario, es decir, con actividad reciente.

El funcionamiento difiere del anterior: cada vez que se escriba un nuevo mensaje en una conversación, para cada usuario que participe en ella se escribirá en la base de datos la fecha en la que se actualizó la conversación. Para saber si ya hemos visitado esta conversación, tendremos el registro de cada visita a las conversaciones en las que el usuario es participante. Así, con sólo una consulta y comparando ambas fechas obtenemos prácticamente la misma información que en el otro modelo de datos ya que el número de mensajes por leer es relevante.

Éste cambio se ha realizado debido a que cada usuario puede ser participante de conversaciones que tendrán mensajes por leer. Si el número de conversaciones en las que participa el usuario es elevado, esta consulta implica una consulta muy costosa a la base de datos lo cual, en Cassandra, no está optimizado.

5.2.3. Archivos e imágenes directamente en la BD

Al estar Cassandra diseñado para manipular grandes cantidades de datos, decidí utilizar ésta característica para que FUFISSANDRA almacenara los archivos de cada usuario y las imágenes de perfil y ofertas en la misma base de datos.

Esta característica tiene una limitación y es que, como mucho, los ficheros (ya sean archivos o imágenes) no deben tener un tamaño mayor a 2 GB, que es el tamaño máximo del campo *value* de una *column*.

En el caso de FUFIFU no es así. Se decidió guardar archivos e imágenes en el sistema de ficheros del servidor para conseguir mayor velocidad a la hora de servirlos a los usuarios.

5.2.4. Búsqueda de ofertas por texto

La aplicación tiene un buscador para filtrar ofertas mediante texto. Éste texto será introducido por el usuario y se buscarán coincidencias con la ruta, el título y la descripción de la oferta.

En la aplicación original se dispone de una tabla que contiene todas las relaciones palabra – identificador de oferta. Así, dado una palabra, obtendremos qué ofertas contienen esa palabra y una puntuación para después clasificarla en los resultados de la búsqueda. Esta relación se genera en el momento de crear o modificar la oferta ya que éste evento no se produce muy a menudo o, por lo menos, menos que la búsqueda. Las palabras se obtienen de la ruta de clasificación, el título y la descripción de cada oferta excluyendo artículos, preposiciones y conjunciones, es decir, guardando sólo las palabras más significativas de la oferta.

Realmente, cuando se crea una oferta, siempre se busca que el título sea lo más descriptivo posible. Basándome en esto, he decidido suprimir la relación palabra – identificador de oferta de las palabras contenidos en la descripción ya que la información relevante que puedan dar es mínima.

6. COMPARACIÓN DE LOS MODELOS DE DATOS

A continuación se hará una comparación de los modelos de datos para tener una idea de cómo se trabaja en cada uno de ellos.

Para ello describiremos la base de datos de cada uno de los modelos.

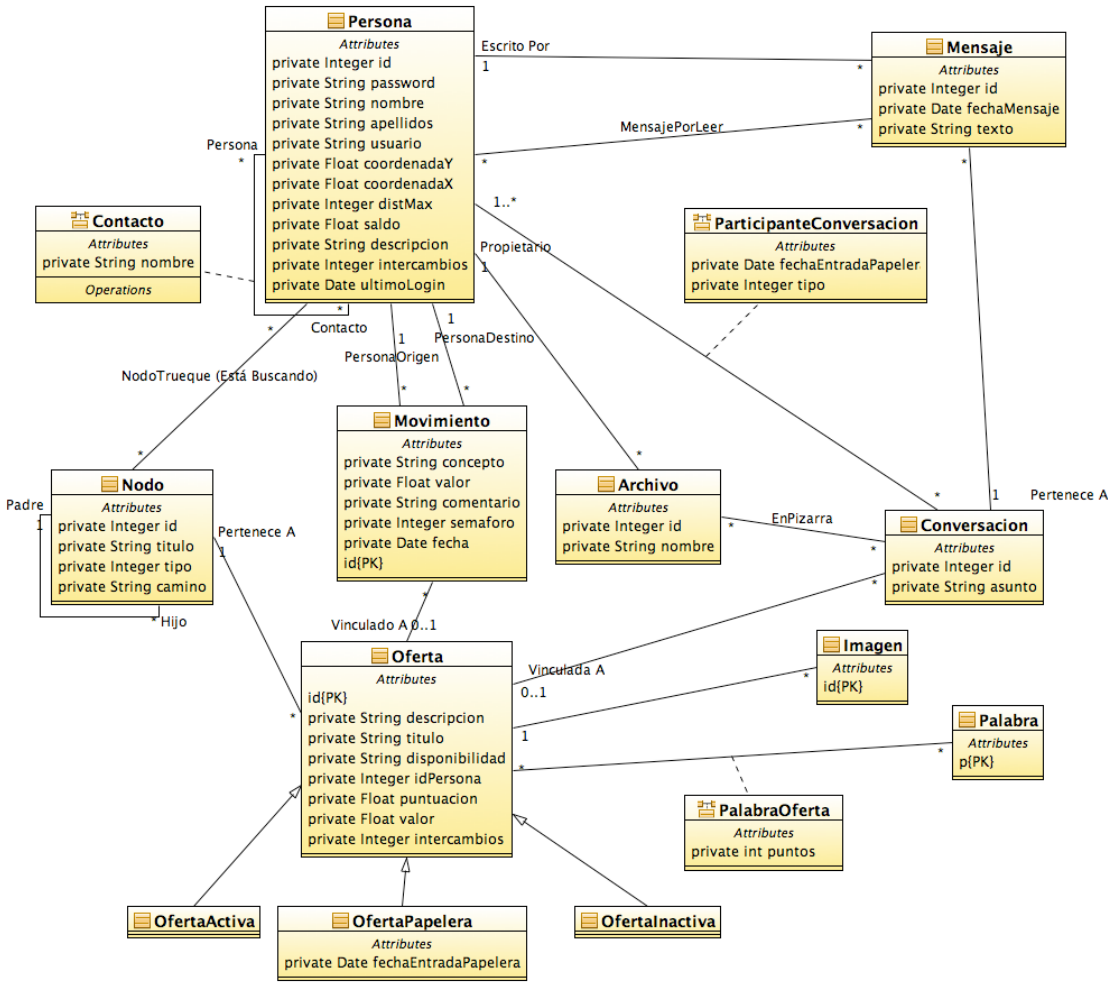
6.1. Modelo de datos de FUFIFU

Está compuesto por 19 tablas y cada una de ellas está relacionada con un objeto de FUFIFU. Estos objetos son una persona, una oferta, una imagen, una conversación, un mensaje, un archivo, un contacto, un nodo o un participante de conversación.

y describiremos cada una de las tablas que forman el modelo de datos.

6.1.1. Diagrama de clases

A continuación, mostraremos el diagrama de clases de FUFIFU



6.1.2. Persona

Ésta tabla contiene la información relacionada con los usuarios de la aplicación. Para cada usuario se guardan los siguientes atributos: identificador de persona, usuario, nombre, apellidos, coordenadas GPS, descripción, número total de intercambios realizados en FUFIFU y distancia máxima por defecto en sus búsquedas.

6.1.3. Oferta

Ésta tabla únicamente contiene dos campos: identificador y tipo. Según el valor del campo tipo, sabremos si una oferta está activa, inactiva o en la papelera.

Para las ofertas, tenemos 3 tablas más que almacenan los datos de cada oferta ya que la tabla Oferta sirve para tenerlas clasificadas.

Estas 3 tablas son OfertaActiva, OfertaInactiva y OfertaPapelera. Tienen los mismos campos: identificador de oferta, identificador del nodo de clasificación al que pertenece la oferta, título de la oferta, descripción de la oferta, identificador de la persona propietaria de la oferta, disponibilidad de dicha persona, valor de la oferta (precio en T's), número de veces que la oferta ha sido intercambiada y puntuación de la oferta.

Además, la tabla OfertaPapelera incorpora un atributo más que nos permite saber desde cuándo una oferta está en la papelera.

6.1.4. Imagen

Dado que las ofertas pueden tener imágenes vinculadas, en la tabla Imagen tenemos los siguientes atributos: identificador de la imagen e identificador de la oferta.

A la hora de decidir cómo guardar las imágenes en el servidor se ha optado por hacerlo directamente en el sistema de ficheros (y no guardando la imagen directamente en binario en la base de datos, como podría haberse hecho), para conseguir mayor velocidad a la hora de servir las imágenes a los usuarios.

6.1.5. Nodo

En la tabla nodo guardaremos el árbol de clasificación de ofertas de FUFIFU.

Cada nodo del árbol tiene un identificador, un título y un identificador del nodo padre.

El árbol de clasificación tendrá dos nodos raíz: TIEMPO y OBJETO. De esta forma, un nodo será de tipo TIEMPO o de tipo OBJETO, dependiendo del nodo raíz del que cuelgue.

6.1.6. Palabra

Guardaremos en esta tabla las palabras que aparecen en el título, ruta de clasificación o descripción de las ofertas de FUFIFU.

6.1.7. PalabraOferta

En esta tabla guardaremos, para cada oferta activa de FUFIFU, qué palabras contiene. Además, a cada palabra de una oferta se le asociará una puntuación que nos permitirá mostrar unos resultados u otros cuando un usuario haga una búsqueda en FUFIFU. Más adelante veremos cómo funcionan las búsquedas en FUFIFU.

6.1.8. Movimiento

FUFIFU permite realizar pagos a sus usuarios y para controlar la persistencia de esta información, es necesaria la tabla Movimiento, donde guardamos el identificador de la persona que realiza el movimiento, el identificador de la persona que recibe el movimiento, el identificador de la oferta vinculada con ese movimiento (si el movimiento fuera una transferencia de T's, no vinculada a ninguna oferta, el valor de éste campo será nulo), el concepto del movimiento, el valor del movimiento (cantidad de T's traspasadas), un comentario y una puntuación (semaforo en la base de datos) donde quien paga pueda plasmar su satisfacción respecto al servicio o bien recibido. La puntuación tiene un valor entero entre 0 y 4.

Además un movimiento tendrá un identificador único y una fecha asociada (indicando cuándo se realizó ese movimiento).

Finalmente, también se guarda el saldo que tendrá la persona que realiza el movimiento y el saldo que tendrá la persona que recibe el movimiento. Así, cuando una persona consulte sus movimientos, el sistema no tendrá que recalcular su saldo acumulado después de cada movimiento.

6.1.9. Conversacion

Una conversación únicamente tiene los siguientes campos: identificador de la conversación, identificador de la oferta vinculada con esa conversación (si la conversación no estuviera vinculada a ninguna oferta, el valor de éste campo sería nulo) y la fecha del último mensaje de la conversación.

6.1.10. ParticipanteConversacion

Dado que las conversaciones pueden tener varios participantes, gracias a esta tabla podremos saber quiénes son los participantes de una conversación. Sus campos son: identificador de conversación, identificador de persona, tipo (que indicará si el

participante tiene esta conversación en estado Activa, Inactiva o Papelera) y fechaEntradaPapelera (que, en caso de que la conversación esté en la papelera, nos indicará desde cuándo está en ella).

6.1.11. Mensaje

Las conversaciones están formadas por mensajes y para cada mensaje guardamos los siguientes datos: identificador del mensaje, identificado de la conversación a la que pertenece, identificador de la persona que lo ha escrito, fecha en la que se ha escrito y texto del mensaje.

De esta forma guardaremos el mensaje en la base de datos una única vez, siendo éste accesible a todos los participantes de la conversación a la que pertenezca el mensaje.

6.1.12. MensajePorLeer

Gracias a esta tabla podemos controlar cuándo un usuario tiene un mensaje por leer. Los campos son simplemente el identificador del mensaje y el identificador de la persona destinataria.

De esta forma, guardaremos en la tabla todas aquellas tuplas (idMensaje, idPersonaDestino) que correspondan a mensajes por leer. Cuando un mensaje sea leído por su destinatario, se eliminará de la tabla MensajePorLeer su tupla correspondiente.

6.1.13. Archivo

Los usuarios tendrán archivos en FUFIFU. Para ello, la tabla Archivo tiene los siguientes atributos: identificador de archivo, identificador de la persona propietaria de ese archivo y nombre del archivo. Los archivos en sí mismos serán guardados directamente en el sistema de ficheros del servidor.

6.1.14. EnPizarra

Los archivos que un usuario tenga en FUFIFU servirán para que pueda colgarlos en “La Pizarra” de cualquiera de sus conversaciones. Empizarra guarda la información relativa a qué archivos están en qué pizarras. Sus atributos son: identificador de archivo e identificador de conversación.

6.1.15. NodoTrueques

Para que FUFIFU pueda calcular qué trueques pueden interesarle a cada usuario, debe conocer aquello que cada usuario de FUFIFU “Está buscando”. La tabla NodoTrueques

guarda aquellos trueques que un usuario está buscando, a través de sus dos atributos: identificador de nodo e identificador de persona.

6.1.16. Contacto

Cualquier usuario de FUFIFU puede tener contactos dentro de la propia plataforma. Para ello, la tabla Contacto tiene tres atributos: el identificador de la persona "propietaria" del contacto, el identificador de la persona contacto y el nombre con el que quiere guardar ese contacto.

6.1.17. Ide

Para controlar que los identificadores de oferta, conversación, mensaje, movimiento, imagen y archivos sean únicos, la plataforma utiliza la tabla Ide.

La tabla Ide tiene los siguientes atributos: tabla y next. Cada vez que se guarda una oferta en la base de datos, FUFIFU consulta en la tabla Ide el valor next correspondiente a la tabla='oferta' y lo incrementa en 1. Así garantizamos que el identificador de cada oferta es único.

De forma análoga se darán de alta nuevos mensajes, movimientos, imágenes o archivos.

6.2. Modelo de datos de FUFISSANDRA

A diferencia del modelo de datos relacional, para diseñar el nuevo modelo hemos de conocer las consultas que se harán a la base de datos a través de la aplicación. Esto es debido a cómo se estructuran los datos en Cassandra.

Antes de mostrar el diagrama de clases, analizaremos las consultas que se realizan a la base de datos actual.

6.2.1 Persona

FUFIFU tiene dos tipos de usuarios: usuarios registrados y usuarios no registrados.

La diferencia entre ambos es que el usuario registrado tiene acceso total a las funcionalidades de la aplicación mientras que el usuario no registrado únicamente puede buscar y visualizar ofertas.

Todo usuario registrado deberá proporcionar sus datos a la aplicación para poder ser identificado dentro de la misma. Además, estos datos pueden ser modificados ya que el usuario podrá cambiar su password, su descripción, su posición GPS o imagen de perfil. Igualmente, otros usuarios pueden consultar sus datos personales para ver las referencias de ofertas intercambiadas o las qué ha registrado.

Para poder entrar a la aplicación como usuario registrado, necesitará el user y password establecidos en el registro.

Por lo tanto, para los datos de una persona, tenemos las siguientes consultas a la base de datos:

- Registro de un usuario
- Consulta de los datos e imagen de perfil
- Intercambios realizados por un usuario
- Modificación de los datos e imagen de perfil
- Acceso a la aplicación

En el modelo relacional, todas estas consultas se realizan a la misma tabla Persona, a excepción de las relacionadas con imagen de perfil e intercambios.

Gracias a que se puede hacer corresponder dos tablas mediante cualquiera de sus atributos, en el modelo relacional basta con una sola tabla para los datos de una persona.

Para la imagen del perfil, no se tiene ninguna tabla ya que ésta se guarda en el sistema de ficheros del servidor con el nombre igual al identificador de persona. Todas las imágenes se almacenan en el mismo directorio.

Los intercambios se obtienen de la tabla Movimiento ya que son movimientos con una oferta asociada.

En cambio, en el modelo no relacional, para realizar las diferentes consultas para obtener los datos de una persona, no podemos utilizar la misma Column Family ya que, debido a su estructura, sólo podemos indexar los datos contenidos en ella por uno de los atributos de la persona.

El atributo más apropiado sería el nombre de usuario ya que éste es único en la aplicación y no existen dos usuarios con el mismo. Pero se ha decidido utilizar el identificador de persona, como en el modelo relacional, porque, además de ser único, no se puede modificar. En cambio, el nombre de usuario sí, lo que implica un control mayor en las referencias a otras Column Families. Además, utilizar el nombre de usuario no resolvería el problema de encontrar una persona por su nombre.

Por lo tanto, se han diseñado tres Column Families: Persona, idPersonaPorUsuario e idPersonaPorNombre.

La imagen de perfil se almacenará directamente en la base de datos gracias a cómo está diseñado Cassandra.

A diferencia del modelo relacional, los intercambios realizados por el usuario se almacenarán separados de los movimientos. Dado que ni los movimientos ni los intercambios se modificarán, podemos tener esta información repetida.

Por lo tanto, tenemos las siguientes Column Families:

- Persona: está indexada por un identificador de persona y contiene todos los datos de registro de esa persona.
- idPersonaPorUsuario: está indexada por el nombre de usuario de una persona y contiene un único identificador de persona (el nombre de usuario es exclusivo).
- idPersonaPorNombre: está indexada por una palabra que será el nombre o cada uno de los apellidos de las persona. Contiene todos los identificadores de persona cuyo nombre o apellidos coincidan con éste índice.
- ImagenPerfil: está indexada por un identificador de persona y contiene la imagen en binario que el usuario haya establecido.
- IntercambiosPorPersona: está indexada por un identificador de persona y contiene cada uno de los intercambios que ha realizado esa persona.

Observaciones:
Podríamos haber aprovechado la característica de Cassandra Secondary Index para tener la relación nombre de usuario-identificador y nombre-identificador, pero debido al número de índices secundarios que se crearían, el rendimiento se vería perjudicado. Esto es debido a que ésta característica está pensada para tener un número pequeño de conjuntos, es decir, si el atributo nombre de usuario o nombre y apellidos tuviera sólo 4 ó 5 opciones.

Gracias a que Cassandra está diseñado para operar con grandes cantidades de datos, tener almacenada la imagen de perfil en la base de datos es una buena idea. De esta manera, nos ahorramos el tener un sistema de gestión para los archivos en el sistema de ficheros del servidor.

6.2.2. Ofertas

Un usuario registrado puede crear ofertas para ofrecer sus habilidades o bienes a la comunidad.

Al crear una oferta, se deben especificar los datos para poder identificarla y, además, ubicarla correctamente en el árbol de clasificación para que los usuarios puedan acceder fácilmente a ella.

Como ocurría con los datos de cada persona también se pueden consultar y modificar, pero, además, es posible que el propietario de una oferta no quiera que otros usuarios accedan a ésta. Para hacer posible esto, cada oferta dispone de un atributo estado que indica si la oferta está activa, inactiva o se desea eliminar. De esta manera, las ofertas estarán clasificadas según éste atributo.

Para clasificar sus ofertas, el usuario necesitará ver todas las que haya creado. Por lo tanto, dispondrá de un apartado o carpeta para cada estado.

Cuando se consulta una oferta, se muestran los últimos intercambios realizados de esta oferta para tener una idea de cómo es la oferta en sí.

Entonces, para los datos de las ofertas tenemos las siguientes consultas:

- Crear una nueva oferta.
- Consulta de los datos e imágenes de una oferta
- Modificación de los datos e imágenes de una oferta
- Intercambios de una oferta
- Consultar todas las ofertas de un usuario según su estado
- Mover ofertas

En el modelo relacional, tenemos varias tablas para poder realizar éstas consultas.

Para la clasificación de las ofertas, existe una tabla Oferta que únicamente contiene la clasificación de la oferta según su estado.

Los datos de las ofertas se guardan en tres tablas según el estado de las mismas: OfertaActiva, OfertaInactiva y OfertaPapelera. Estas mismas tablas se utilizarán para obtener todas las ofertas de un usuario ya que éste las consultará según un estado dado.

Para conocer qué imágenes tiene cada oferta, se consultará la tabla Imagen.

Igual que pasaba con los intercambios de cada persona, los intercambios de las ofertas se obtienen de los movimientos realizados con esta oferta.

En el modelo no relacional, he condensado prácticamente todas estas tablas en una única Column Family, excepto la tabla Imagen que tiene dedicada una Column Family para almacenarlas.

Para consultar todas las ofertas de un usuario, he decidido crear una Column Family para éste fin. Para no necesitar un control de los datos y tenerlos duplicados en varias Column Families, se almacenarán los identificadores de las ofertas de cada persona para, posteriormente, obtener la información de cada una de ellas.

Aprovechando la característica Secondary Index, se ha establecido el atributo estado de las ofertas como segundo índice. Así, al buscar una oferta, sabremos si está activa o no sin necesidad de consultar ésta información relacionando al propietario, esto es, sin necesidad de utilizar la Column Family descrita en el párrafo anterior.

La Column Family que alberga las imágenes de las ofertas, será similar a la que se utiliza para almacenar la imagen de perfil de cada usuario sustituyendo el identificador de persona por el identificador de oferta. Se podría haber incorporado como un atributo más de la Column Family Ofertas, pero debido a que se realizan más consultas a los datos que a la propia imagen, es mejor opción tenerlo separado.

Para los intercambios, al igual que pasaba con los de cada persona, se ha creado una Column Family para almacenarlos.

Por lo tanto, las Column Families que he diseñado son:

- Ofertas: está indexada por un identificador de oferta y contiene los datos de esa oferta.
- OfertasPorPersona: es de tipo Super y está indexada por el identificador de persona y por el estado de las ofertas. Contiene los identificadores de las ofertas que se encuentren en ese estado y de esa persona.
- ImagenesPorOferta: está indexada por un identificador de oferta y contiene cada todas las imágenes en binario de esa oferta.
- IntercambiosPorOferta: está indexada por un identificador de oferta y contiene todos los intercambios realizados con esta oferta.

6.2.3. Búsqueda de ofertas

FUFIFU permite realizar búsquedas de ofertas de dos maneras: por tipo de oferta y por palabras clave.

Las palabras clave se establecen a partir de la ruta de clasificación, el título y la descripción de la oferta. De esta manera, al crear o modificar una oferta, se generarán de nuevo las palabras clave.

Por lo tanto, para buscar ofertas únicamente tendremos dos consultas:

- Obtener ofertas de un tipo determinado.
- Obtener ofertas que contengan en la ruta de clasificación, el título o la descripción un cierto texto.

En el modelo relacional, tenemos una tabla para cada consulta.

Las búsquedas por tipo de oferta se realizan sobre la tabla Oferta indexando por el tipo de cada oferta según se haya seleccionado.

Las búsquedas por palabras clave se realizan sobre la tabla PalabraOferta. Cada fila de esta tabla contiene una palabra, un identificador de oferta y una puntuación que indica cómo de relevante es esta palabra para esa oferta.

En el modelo no relacional, he optado por seguir la misma estructura que en el modelo original.

Las búsquedas por tipo se podrían realizar estableciendo el atributo tipo como Secondary Index ya que cumple las condiciones necesarios para no perjudicar al rendimiento. Pero, debido a operaciones posteriores para ordenar las ofertas obtenidas, he preferido diseñar una Column Family dedica a ésta consulta.

La mejor opción para las búsquedas por palabra es la optada en el modelo relacional.

Por lo tanto, las Column Families creadas para satisfacer estas consultas son:

- OfertasPorRuta: es del tipo Super y está indexada por cada uno de los tipos de oferta. Dentro de cada Super Column, tendremos un conjunto de rows indexadas por cada una de las rutas de ese tipo y cada row almacena los identificadores de oferta que se hayan creado con esa ruta.
- OfertasPorPalabra: está indexada por una palabra y cada row contiene los identificadores de oferta cuya clasificación y cuyo título contengan esta palabra.

Observación:
La puntuación para la búsqueda por palabras se realizará una vez obtenida la lista con esas palabras ya que, al reducir el número de campos de los que se obtiene, no supone un coste elevado calcularlo.

6.2.4. Conversaciones

Como cualquier comunidad hoy en día, FUFIFU permite establecer conversaciones entre grupos de usuarios.

Las conversaciones están formadas por participantes, mensajes e, incluso, archivos.

Para evitar que un usuario que no participa en una conversación pueda leer su contenido, necesitamos una lista con los usuarios asociados a esa conversación.

Cada participante tiene clasificadas las conversaciones en las que participa en tres carpetas: activas, inactivas y en papelera. Según se encuentre la conversación, el usuario recibirá notificaciones de mensajes escritos o no. Igual que ocurría con las ofertas, el usuario verá todas sus conversaciones en estos tres apartados, pudiendo moverlas entre ellos.

Las conversaciones permiten a los usuarios subir archivos para compartir con los demás participantes. Entonces, para cada conversación se deberá tener una lista con los archivos subidos por los participantes.

Las consultas que se realizan sobre estos datos son:

- Crear conversaciones
- Crear mensajes
- Obtener las conversaciones que participa un usuario
- Consultar una conversación (u obtener todos los mensajes de una conversación)
- Añadir participantes a una conversación
- Obtener una lista de participantes de una conversación
- Subir archivos a una conversación
- Obtener los archivos subidos a una conversación
- Mover conversaciones

En el modelo relacional, se necesitan seis tablas para poder satisfacer estas consultas.

Los datos de las conversaciones se almacenan en la tabla Conversacion.

Para obtener las conversaciones en las que participa un usuario, se utiliza la tabla ParticipanteConversacion.

Para los datos de los mensajes, se utilizan dos tablas: Mensaje y MensajePorLeer. Mensaje contiene los datos de cada mensaje que se haya escrito en una conversación. MensajePorLeer relaciona un mensaje que no se haya leído con cada uno de los participantes de una conversación hasta que éste lo ha ido.

Para la gestión de archivos, se utilizan las tablas Archivo y EnPizarra. Archivo contiene la relación archivo-propietario y EnPizarra la relación archivo-conversación.

En el modelo no relacional el esquema es muy similar, pero se ha cambiado la notificación de los mensajes sin leer, como se ha explicado en el capítulo anterior.

El cambio se ha realizado porque Cassandra relaja el tema de la consistencia para obtener una mayor velocidad de respuesta. Al ser los mensajes por leer un contador que se incrementa y decrementa muy frecuentemente, se vería afectado el sistema de notificaciones de mensajes.

Las Column Families creadas para realizar estas consultas son:

- Conversaciones: está indexada por un identificador de conversación y contiene un resumen de esa conversación indicando la fecha del último mensaje escrito, el asunto y el identificador de oferta asociada si lo tuviera.
- MensajesPorConversación: está indexada por un identificador de conversación y contiene los datos de todos los mensajes que se hayan escrito en esa conversación.
- ConversacionesPorPersona: está indexada por un identificador de persona y un estado. Cada row contiene los identificadores de las conversaciones en las que participa esa persona y que tiene clasificada por ese estado.

- ParticipantesPorConversación: está indexada por un identificador de conversación y contiene los identificadores de persona que pueden leer la conversación.
- ConvActRecientePorPersona: su índice es un identificador de persona y contiene los identificadores de conversación con mensajes nuevos que aún no haya leído ese usuario.
- UltimaVisitaConversacionPorPersona: está indexada por un identificador de persona y contiene la fecha en la que se visitaron cada una de las conversaciones en las que participa.

6.2.5. Archivos

Como he comentado en el apartado anterior, FUFIFU nos permite subir archivos para compartir en conversaciones.

Para llevar un control de estos archivos, cada usuario dispone de un apartado con todos los datos de los archivos subidos: qué archivo se ha subido y en qué conversación se está compartiendo.

Por lo tanto, las consultas que se hacen sobre estos datos son:

- Archivos subidos de un usuario
- Conversaciones en las que se ha compartido un archivo

En el modelo relacional, se utilizan las tablas antes mencionadas Archivo y EnPizarra.

En el modelo no relacional, se requieren dos Column Families: ArchivosPorPersona y ConversacionesPorArchivo.

El motivo por el que se ha creado la Column Family ArchivosPorPersona es el mismo por el que se han creado las otras Column Families para almacenar la relación objeto-persona.

Igual pasa con ConversacionesPorArchivo, pero cambiando la relación por conversación-archivo.

Por lo tanto, las Column Families creadas son:

- ArchivosPorPersona: es del tipo Super y está indexada por un identificador de persona y por un identificador de archivo. Contiene cada row el nombre del archivo y el archivo en binario que, al igual que las imágenes, se ha decidido almacenar directamente en la base de datos.
- ConversacionesPorArchivo: está indexada por un identificador de archivo y contiene todos los identificadores de conversación en las que se haya compartido.

6.2.6. Movimientos

Para que quede constancia de los traspasos de T's que realizan los usuarios, es necesario tener un registro de éstos.

Estos movimientos son privados para ambos usuarios que los realizan. Así que las consultas que se harán sobre estos datos serán sencillas:

- Obtener movimientos de un usuario entre dos fechas

En el modelo relacional se utiliza la tabla Movimiento, la cual contiene todos los datos de esta trasferencia.

Dado que estos movimientos se consultan entre fechas, en el modelo no relacional he aprovechado la estructura Super Column Family para tenerlos ya ordenados.

Como ha pasado en los casos anteriores, tendremos separados los datos de los movimientos y los usuarios implicados, es decir, tendremos dos Column Families.

Por lo tanto, las Column Families que se han creado para realizar esta consulta son:

- Movimientos: está indexada por el identificador de movimiento y contiene los datos relacionados con éstos.
- MovimientosPorPersona: es del tipo Super y está indexada por un identificador de persona y una fecha. Contiene los identificadores de los movimientos realizados por esa persona ya ordenados por la hora, aprovechando el comparador configurable de las Columns.

6.2.7. Contactos

Los usuarios disponen de una agenda de contactos para guardar usuarios con los que iniciar una conversación o, simplemente, para tener un acceso directo a sus ofertas.

Por lo tanto, las consultas para gestionar estos datos son:

- Agregar contacto
- Obtener contactos
- Eliminar contacto

En el modelo relacional, todo se gestiona a partir de la tabla Contacto, la cual contiene la relación entre ambos usuarios.

El esquema en el modelo no relacional es idéntico. Se dispone de una Column Family para tratar esta relación.

La Column Family asociada a estas consultas es:

- ContactosPorPersona: está indexada por un identificador de persona y contiene los identificadores de las personas que son contactos de ésta.

6.2.8. Trueques

FUFIFU dispone de una función muy útil y es que sugiere ofertas a los usuarios según sus preferencias.

Para gestionar estos trueques son necesarias estas consultas:

- ¿Qué busca un usuario?
- Obtener posibles trueques para un usuario

En el mundo relacional, esta operación se realizaba a través de identificadores de los nodos del árbol de clasificación.

En cambio, en el modelo no relacional, he optado por substituir el identificador del nodo por su ruta.

De esta manera, al tener las ofertas ordenadas por rutas, en una sola consulta obtendremos todos los identificadores de ofertas que puedan interesar al usuario.

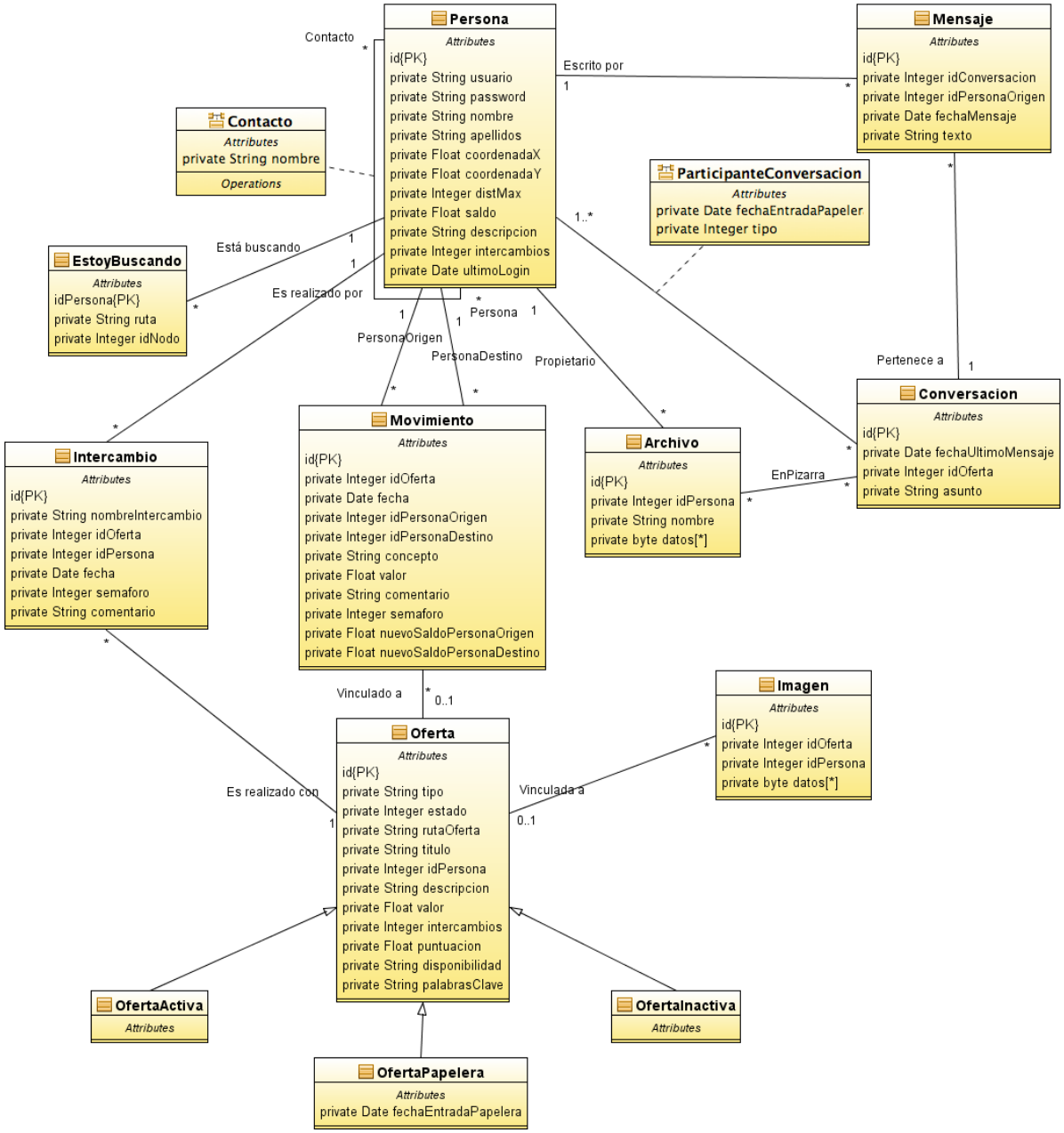
Por lo tanto, las Column Family creadas para estas consultas son.

- Buscando: está indexada por un identificador de persona y contiene las rutas de las ofertas que está buscando
- OfertasPorRuta: es del tipo Super y está indexada por los tipos de las ofertas (OBJETO o TIEMPO) y por su ruta. Contiene cada row los identificadores de oferta con esa ruta en el árbol de clasificación.

Observaciones:
Después de diseñar el modelo de datos, tuve que añadir los identificadores de los nodos porque la función que dibuja el árbol de clasificación estaba diseñada con éste atributo. Igualmente, los identificadores de nodo son sólo utilizados para dibujar el árbol y no para la gestión de lo que está buscando un usuario.

6.3.1. Diagrama de clases

A continuación se mostrará el diagrama de clases. Es muy similar al de FUFIFU, pero no tiene la clase Palabra ni la clase Nodo y se le ha añadido la clase Intercambio y EstoyBuscando. Además, a las clases Archivo e Imagen se les ha añadido el atributo datos.



7. PRUEBAS DE RENDIMIENTO

7.1. Objetivos de la evaluación

El objetivo de la evaluación es experimentar si realmente el modelo no relacional suple las carencias del modelo relacional, es decir, si mejora el rendimiento en un sistema altamente escalable, como lo es una red social, con el nuevo modelo de datos.

7.2. Herramientas

Para llevar a cabo estos experimentos, hemos utilizado varias herramientas que a continuación describiremos.

7.2.1. Aplicaciones web FUFIFU y FUFISSANDRA

Obviamente necesitaremos ambas aplicaciones web para evaluar el rendimiento de ambos modelos.

Como ya hemos descrito en puntos anteriores, hay diferencias en las aplicaciones, pero en las pruebas que realizaremos sobre ellas se utilizarán los mismos métodos y lo único que cambiará será el acceso a la base de datos.

7.2.2. Servidor web con soporte de servlets TOMCAT

Para que funcionen las aplicaciones web, necesitaremos un servidor web con soporte de servlets. Éste es Apache Tomcat.

Utilizaremos la versión 6.0.29 ya que, inicialmente, FUFIFU se desarrolló con ésta y, tras comprobar su funcionamiento, FUFISSANDRA también trabaja correctamente con esta versión.

Lo configuraremos con el usuario ahio para la versión FUFIFU para poder acceder a la base de datos de MySQL. En el caso de FUFISSANDRA no hace falta configurar ningún usuario para el acceso a la base de datos.

7.2.3. Sistema gestor de base de datos

Para la gestión de las bases de datos utilizaremos MySQL para FUFIFU y Cassandra para FUFISSANDRA

7.2.3.1. MySQL

Utilizaremos la versión 5.1.41-3ubuntu12.10 de los repositorios de Ubuntu. Únicamente deberemos crear el usuario mencionado anteriormente.

7.2.3.2. Cassandra

Utilizaremos la versión 0.7.0. Decidimos no actualizar a la versión más reciente ya que está en continuos cambios y actualizaciones.

Actualmente se dispone de la versión 1.0.6 en la cual se han arreglado varios fallos encontrados en versiones anteriores.

Con la versión 0.7.0, FUFISSANDRA trabaja correctamente.

7.2.4. Scripts de ejecución

Para ejecutar las pruebas de rendimiento he elaborado un script que define las bases de datos de ambos modelos y genera los estados iniciales para cada prueba. Los datos generados son: nombres y apellidos y coordenadas GPS de los usuarios y el número de ofertas, movimientos y rutas para trueques de cada usuario. En el apartado 7.3 se describen más detalladamente estos datos.

Para cada operación se ha creado un script que, mediante el comando curl de UNIX, se conectará al servlet de la aplicación y enviará los parámetros.

7.2.5. Procesado de datos y gráficas

Una vez acabadas las pruebas de rendimiento, deberemos procesar los datos para su posterior representación. Éste procesado consiste en calcular las medias de cada operación teniendo en cuenta que tenemos los resultados de los accesos a la base de datos y el tiempo total que tarda el método en devolver los datos solicitados. Por lo tanto, los resultados que se mostrarán en las gráficas son las medias de cada acceso a la base de datos para ese método y la media de cada petición al método.

Para calcular las medias de los valores y elaborar las gráficas para visualizar mejor los resultados, he utilizado la herramienta de cálculo de Microsoft Excel 2010.

7.3. Descripción de las pruebas

Las pruebas de rendimiento se reparten en tres bloques: escritura, lectura y combinación de ambas.

En el bloque de escritura se medirá el tiempo que tardan en realizarse las operaciones de añadir nueva persona, nueva oferta, rutas para posibles trueques y realizar pagos entre usuarios.

En el bloque de lectura se medirá el tiempo que tardan en realizarse las operaciones de búsqueda de ofertas por palabras y por el tipo de oferta, posibles trueques y movimientos entre las fechas señaladas (varían según la prueba).

Dadas las diferentes implementaciones entre los modelos de datos, se compararán las siguientes características:

- *Secondary Index* de Cassandra para las búsquedas por tipo de oferta.
- *RangeSlice* de Cassandra para las búsquedas de movimientos.

Aprovecharemos éstas características del modelo de datos de Cassandra para comprobar si mejora el rendimiento de la aplicación.

En el bloque de combinación de lectura y escritura se medirá el tiempo que tardan en realizarse las operaciones de leer las rutas de oferta para posibles trueques que ha añadido el usuario inicialmente y añadir las restantes para tener todas las posibles rutas; obtener los títulos de las ofertas y modificarlos para generar de nuevo las palabras clave a buscar; y, finalmente, mover las ofertas de activas a inactivas.

Como eliminar toda la base de datos acabado un estado supondría un coste muy elevado en tiempo, se comprimirán con la herramienta NODETOOL COMPRESS para Cassandra y la sentencia OPTIMIZE TABLE para MySQL. Además también utilizaremos la herramienta NODETOOL CLEANUP para que se borren las *keys* que hayan en caché.

7.3.1. Equipo y estados de la base de datos

El equipo utilizado es un portátil AHTEC SENSE FL92 con las siguientes características:

- Placa base Compal JFL92
- Procesador Intel Core 2 Duo T9300 @ 2,5 GHz
- Memoria RAM de 4 GBytes DDR2-667 SDRAM
- Disco duro Toshiba MK3252GSX SATA-II
- Sistema Operativo Ubuntu 10.04 32 bits

A continuación se definen los 3 estados iniciales de la base de datos y las operaciones que se realizarán para obtener los resultados para poder comparar el rendimiento ambos modelos de datos.

Estado 1

El número de usuarios será 100 y cada uno de ellos tendrá, de media, 2 ofertas, 1 ruta de oferta seleccionada para posibles trueques y habrá hecho 1,5 movimientos siendo él el emisor del pago entre 3 días.

Para éste estado se realizarán las siguientes operaciones:

- Escritura
  Añadiremos 50 usuarios más.
  Aumentaremos las medias de los usuarios de:
  -- Ofertas a 3
  -- Rutas de oferta seleccionadas para posibles trueques a 1,5
  -- Movimientos a 2

- Lectura
  Buscaremos ofertas por las 19 palabras clave más utilizadas, extraídas de los títulos de las ofertas, y por los dos tipos de ofertas que hay. Las búsquedas por tipo se efectuarán 50 veces por cada tipo.
  Buscar posibles trueques para todos los usuarios.
  Buscaremos los movimientos para todos los usuarios.
- Combinación de lectura y escritura
  Añadiremos las rutas restantes para cada usuario para posibles trueques.
  Modificaremos los títulos de todas las ofertas.
  Moveremos todas las ofertas de activas a inactivas de todos los usuarios.

Estado 2

El número de usuarios será 1000 y cada uno de ellos tendrá, de media, 4 ofertas, 3 rutas de oferta seleccionadas para posibles trueques y habrá hecho 4 movimientos siendo él el emisor del pago entre 8 días.

Para éste estado se realizarán las siguientes operaciones:

- Escritura
  Añadiremos 800 usuarios más.
  Aumentaremos las medias de los usuarios de:
  -- Ofertas a 6
  -- Rutas de oferta seleccionadas para posibles trueques a 4
  -- Movimientos a 6

- Lectura
  Buscaremos ofertas por las 19 palabras clave más utilizadas, extraídas de los títulos de las ofertas, y por los dos tipos de ofertas que hay. Las búsquedas por tipo se efectuarán 50 veces por cada tipo.
  Buscar posibles trueques para todos los usuarios.
  Buscaremos los movimientos para todos los usuarios.

- Combinación de lectura y escritura
  Añadiremos las rutas restantes para cada usuario para posibles trueques.
  Modificaremos los títulos de todas las ofertas.
  Moveremos todas las ofertas de activas a inactivas de todos los usuarios.

Estado 3

El número de usuarios será 10000 y cada uno de ellos tendrá, de media, 5 ofertas, 3 rutas de oferta seleccionadas para posibles trueques y habrá hecho 6 movimientos siendo él el emisor del pago entre 13 días.

Para éste estado se realizarán las siguientes operaciones:

- Escritura
  Añadiremos 10000 usuarios más.
  Aumentaremos las medias de los usuarios de:
  -- Ofertas a 7
  -- Rutas de oferta seleccionadas para posibles trueques a 5
  -- Movimientos a 8

- Lectura
  Buscaremos ofertas por las 19 palabras clave más utilizadas, extraídas de los títulos de las ofertas, y por los dos tipos de ofertas que hay. Las búsquedas por tipo se efectuarán 50 veces por cada tipo.
  Buscar posibles trueques para todos los usuarios.
  Buscaremos los movimientos para todos los usuarios.

- Combinación de lectura y escritura
  Añadiremos las rutas restantes para cada usuario para posibles trueques.
  Modificaremos los títulos de todas las ofertas.
  Moveremos todas las ofertas de activas a inactivas de todos los usuarios.

7.4. Resultados

Una vez han concluido las pruebas de rendimiento, pasamos al análisis de los resultados.

Para cada operación y estado, se mostrarán unas gráficas con el tiempo medio por acceso a la base de datos que se ha realizado en cada método y la media del tiempo que se ha necesitado para devolver el dato al usuario.

A continuación se muestran las gráficas de los resultados según la operación realizada: escritura, lectura y combinación de éstas.

7.4.1. Escritura

Antes de mostrar las gráficas con los resultados, describiremos los accesos que se realizan a la base de datos.

Registro de una persona

En el caso de Cassandra, tenemos que realizar cinco accesos: uno para almacenar los datos de la persona, otro para almacenar la relación persona con su nombre de usuario y otros tres para almacenar la relación persona con su nombre y sus dos apellidos. Todos ellos se realizan sobre Column Family.

En el caso de MySQL, con un solo acceso para almacenar los datos de la persona es suficiente.

Registro de una oferta

En Cassandra, tenemos que realizar cuatro accesos: uno para guardar los datos de la oferta, otro para la relación oferta con la persona propietaria, otro para almacenar la relación ruta de la oferta con la oferta y otro para almacenar cada relación palabra clave con oferta. De los cuales dos accesos a Column Family y otros dos a Super Column Family.

En MySQL, se realizan tres accesos: uno para guardar el estado de la oferta, otro para guardar los datos de la oferta y otro por cada palabra clave establecida.

Establecer rutas para posibles trueques

En Cassandra, esta operación implica un único acceso para almacenar la ruta de ofertas para posibles trueques y se realiza a una Column Family.

Igual que en Cassandra, en MySQL también se realiza un único acceso, pero en vez de almacenar la ruta se almacena el identificador de nodo.

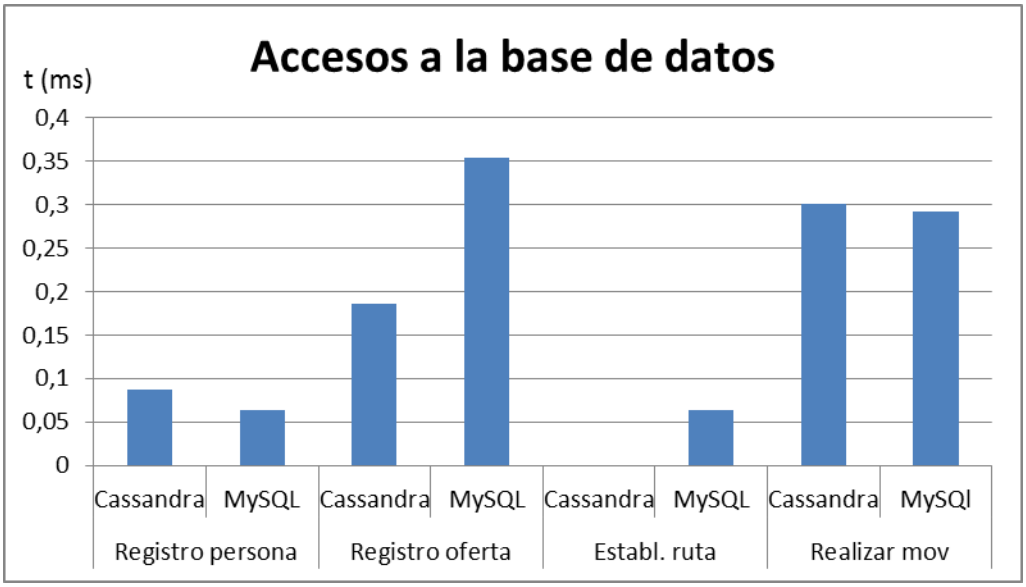
Realizar movimiento

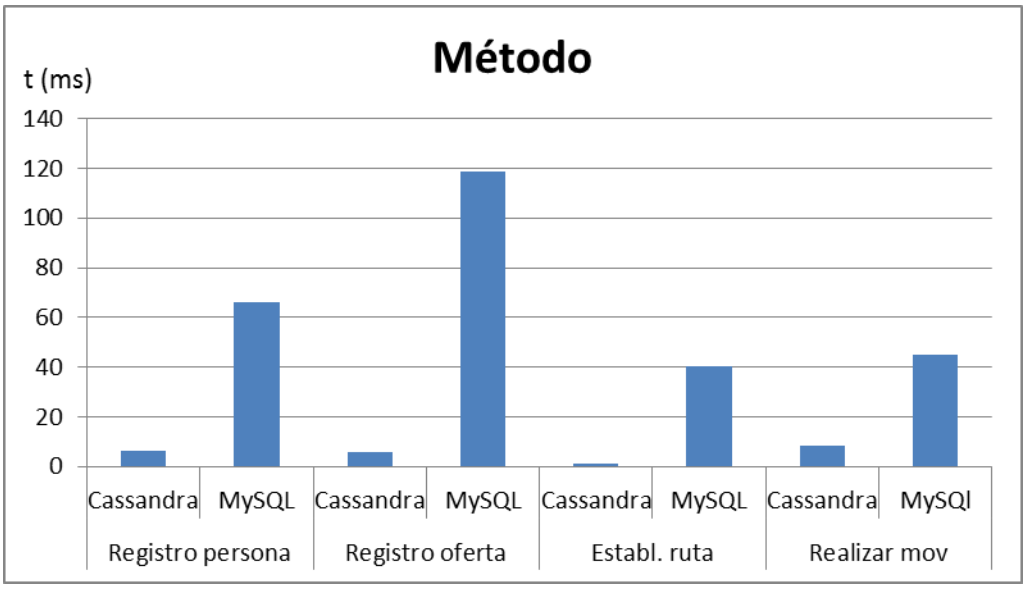
En Cassandra, se realiza un acceso para almacenar los datos del movimiento, dos para guardar la relación movimiento con las personas implicadas, otro para guardar el intercambio y dos más para actualizar los estados de las personas. En total son seis accesos, los cuales cuatro son a una Column Family y dos a Super Column Family.

En MySQL, se realiza un acceso para almacenar los datos del movimiento y dos accesos para guardar la relación movimiento con las personas implicadas. En total son tres accesos.

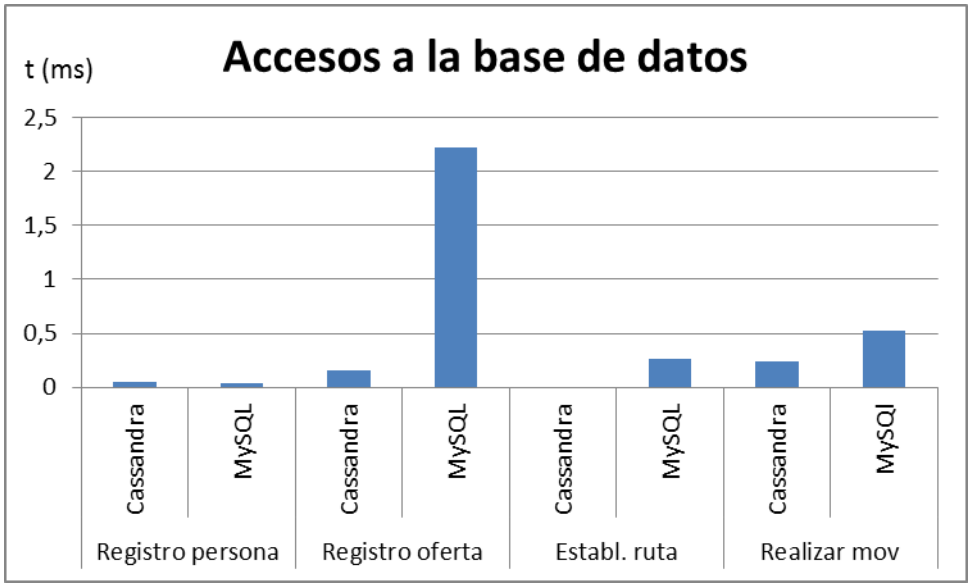
Gráficas

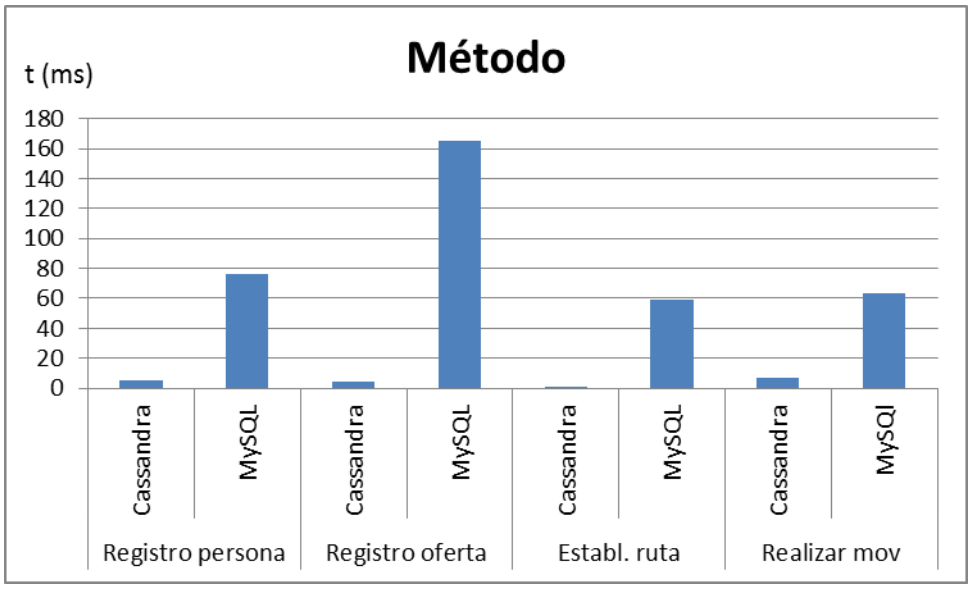
Estado 1



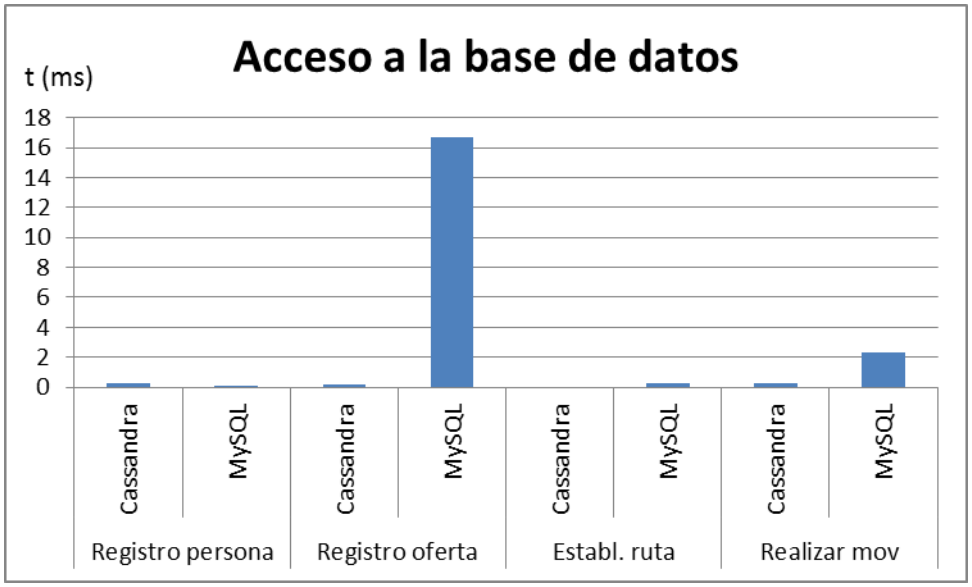


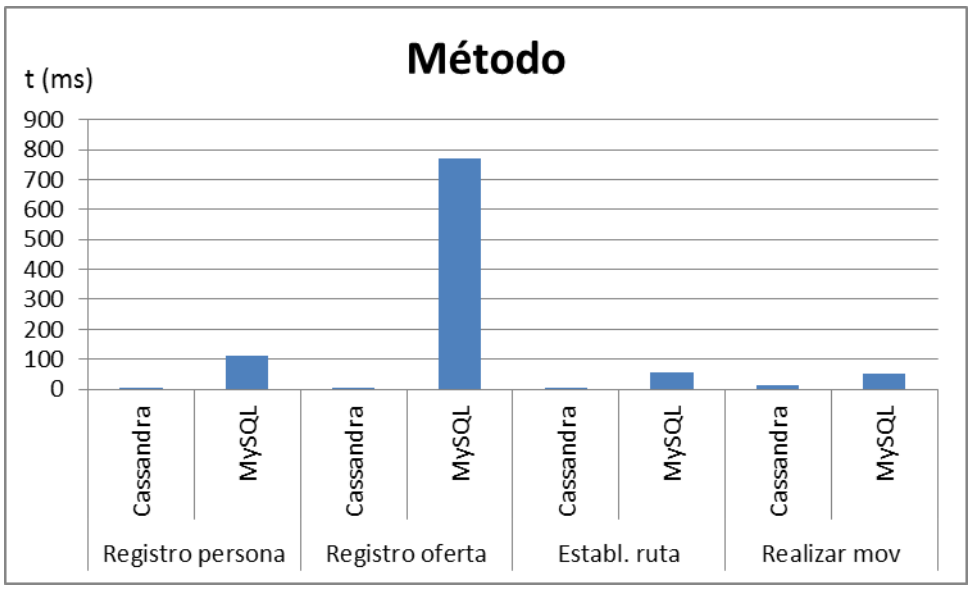
Estado 2


Accesos a la base de datos
t (ms)
2,5
2
1,5
1
0,5
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQl
Registro persona
Registro oferta
Establ. ruta
Realizar mov


Método
t (ms)
180
160
140
120
100
80
60
40
20
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQl
Registro persona
Registro oferta
Establ. ruta
Realizar mov

Estado 3


Acceso a la base de datos
t (ms)
18
16
14
12
10
8
6
4
2
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Registro persona
Registro oferta
Establ. ruta
Realizar mov


Método
t (ms)
900
800
700
600
500
400
300
200
100
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Registro persona
Registro oferta
Establ. ruta
Realizar mov

7.4.2. Lectura

A continuación, describiremos los accesos que se realizan a la base de datos para los métodos que hemos medido.

Búsqueda por palabra

En el caso de Cassandra, implica cuatro accesos: uno para obtener los identificadores de las ofertas que contengan esa palabra en sus palabras clave, otro para obtener los datos de cada oferta obtenida y dos más para obtener los identificadores de nodo de la ruta de las ofertas para, posteriormente, dibujar el árbol de clasificación. Todos ellos se hacen a Column Family.

En el caso de MySQL, implica, también, cuatro accesos: dos para obtener las ofertas que contengan esa palabra en sus palabras clave y otros dos para obtener el identificador del nodo padre e hijo.

Búsqueda por tipo de oferta

Es idéntica a la anterior. Lo único que para obtener las ofertas, habrá que filtrar por el tipo de cada una y no por sus palabras clave.

En Cassandra se utilizará la Super Column Family OfertasPorRuta y en MySQL se buscará en la misma tabla OfertaActiva.

En MySQL se requieren dos accesos a la base de datos: uno para obtener las ofertas activas de ese tipo y otro para obtener los identificadores de los nodos a los que pertenecen las ofertas para dibujar el árbol de clasificación que se le mostrará al usuario.

Consulta de movimientos

En Cassandra, se necesitan dos accesos: uno para obtener los identificadores entre las fechas dadas y otro para obtener los datos de los mismos. El primer acceso es a una Super Column Family y el segundo a una Column Family.

En MySQL, se necesita un único acceso ya que se obtienen todos los datos de una única tabla.

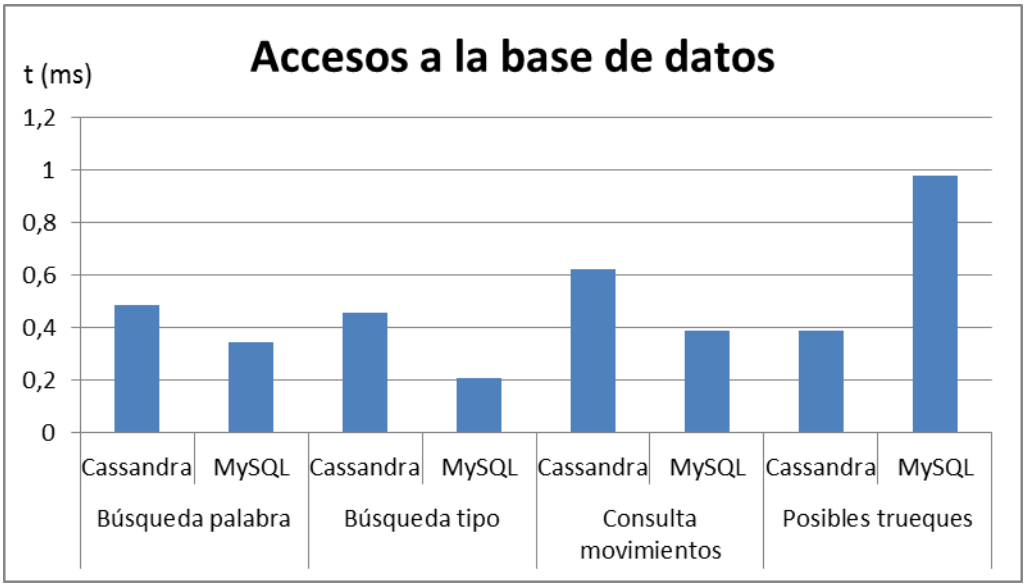
Consultar posibles trueques

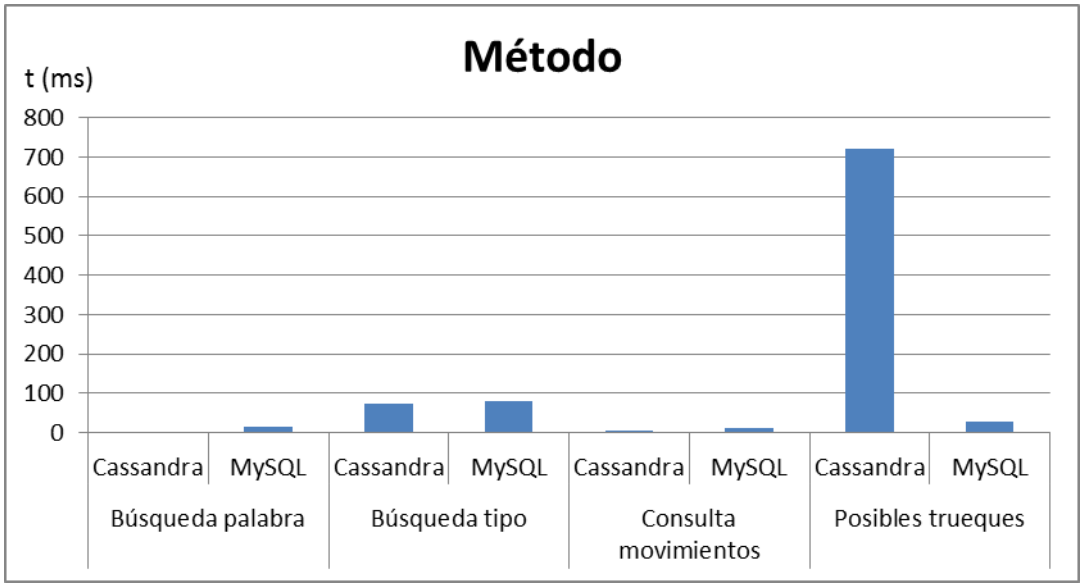
En Cassandra necesitamos tres accesos: uno para obtener las rutas que busca el usuario, otro para obtener los identificadores de las ofertas de cada ruta y otro para obtener los datos de cada una de las ofertas. Las dos primeras se realizan sobre Super Column Families y la última sobre la Column Family.

En MySQL son necesarios cuatro accesos: dos accesos para los identificadores de los nodos de los usuarios que puedan hacer un trueque y otros dos para cada oferta.

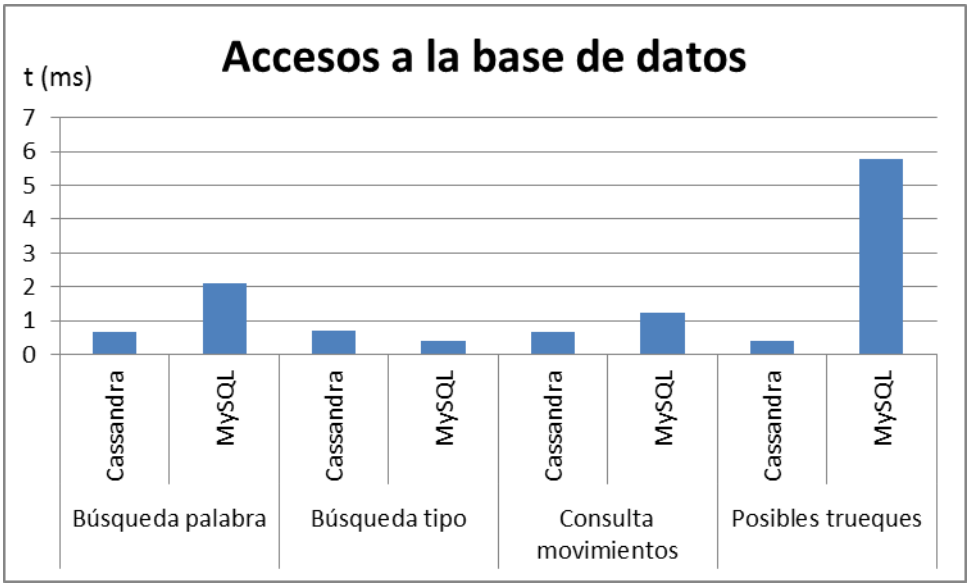
Gráficas

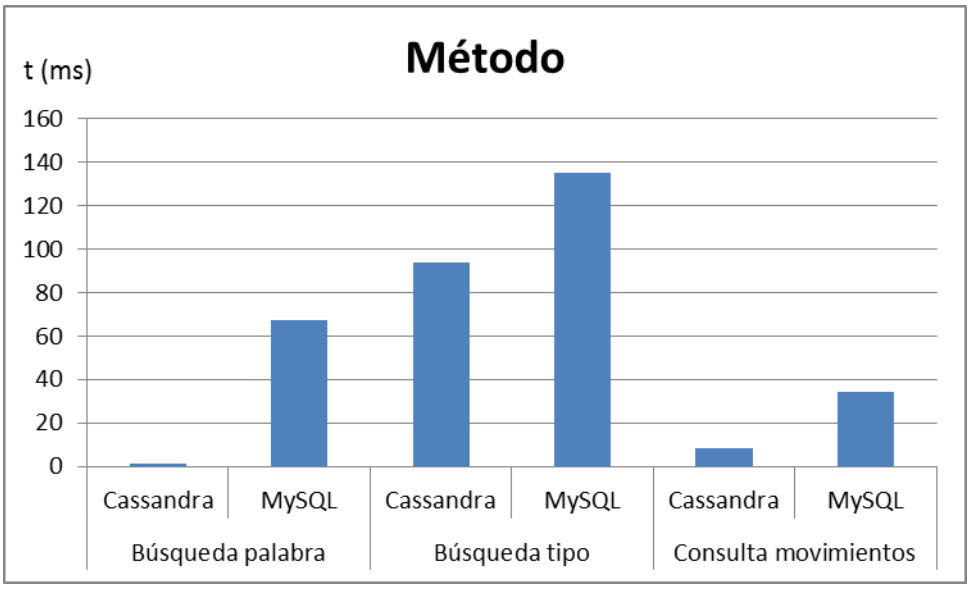
Estado 1

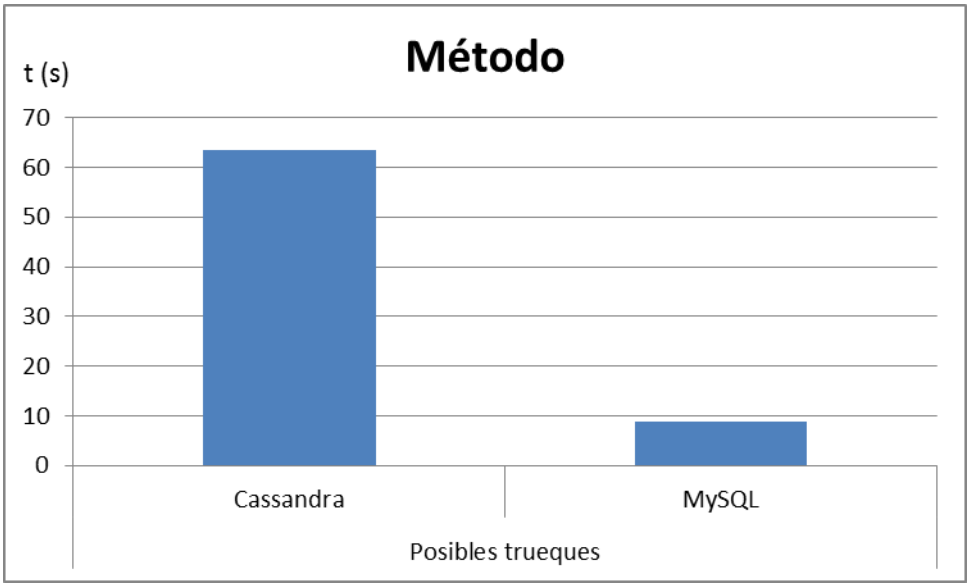




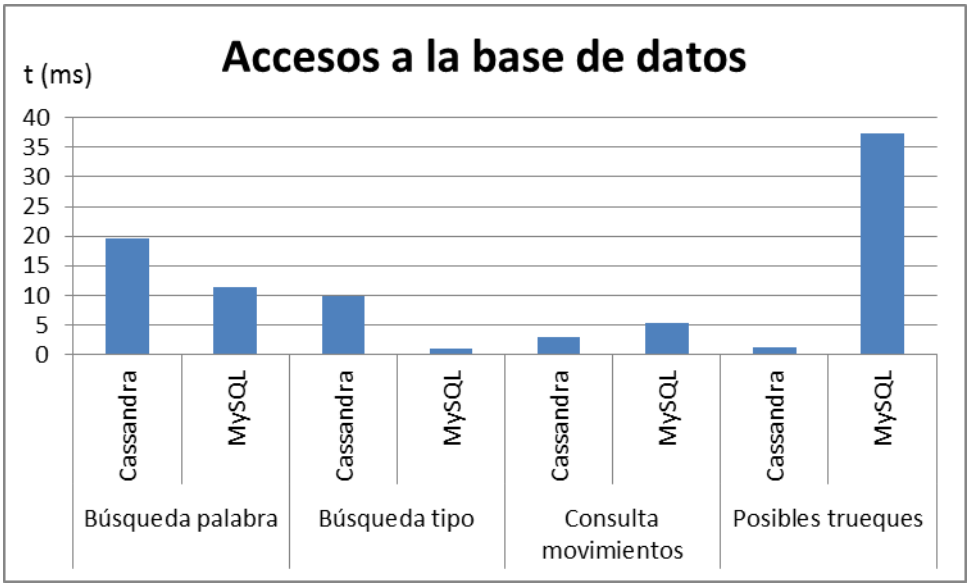
Estado 2

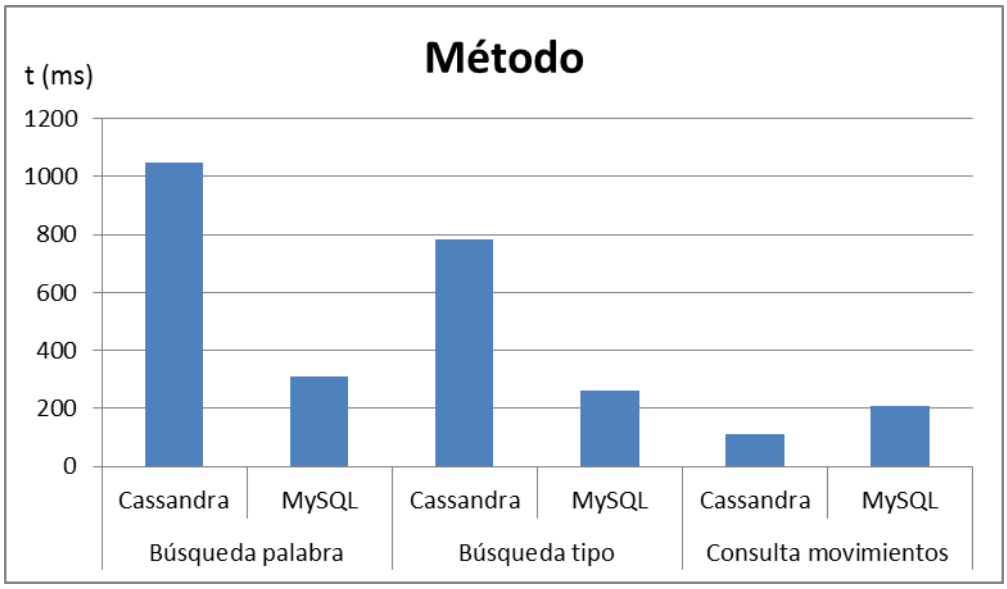

Accesos a la base de datos
t (ms)
7
6
5
4
3
2
1
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Búsqueda palabra
Búsqueda tipo
Consulta movimientos
Posibles trueques

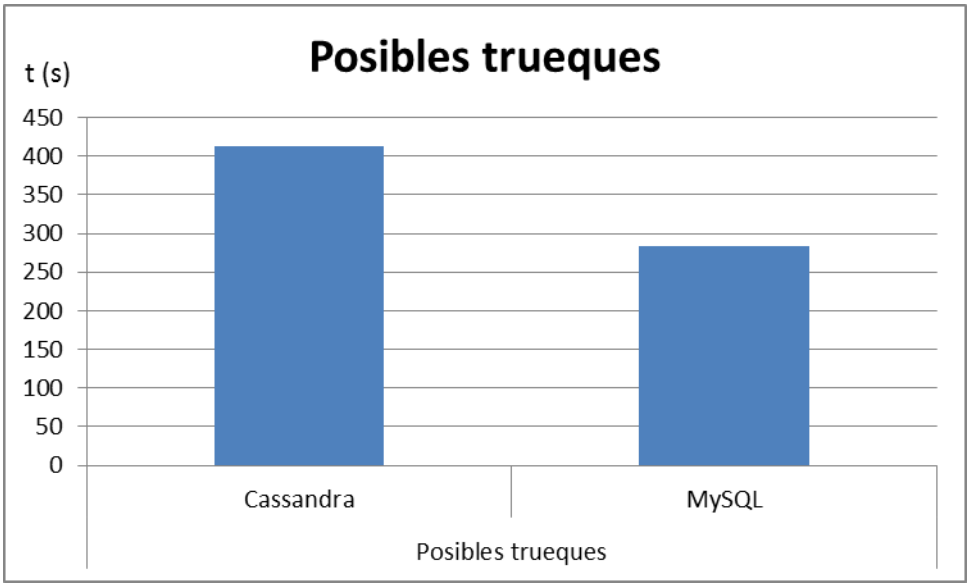

Método
t (ms)
160
140
120
100
80
60
40
20
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Búsqueda palabra
Búsqueda tipo
Consulta movimientos


Método
t (s)
70
60
50
40
30
20
10
0
Cassandra
MySQL
Posibles trueques

Estado 3


Accesos a la base de datos
t (ms)
40
35
30
25
20
15
10
5
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Búsqueda palabra
Búsqueda tipo
Consulta movimientos
Posibles trueques


Método
t (ms)
1200
1000
800
600
400
200
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Búsqueda palabra
Búsqueda tipo
Consulta movimientos


Posibles trueques
t (s)
450
400
350
300
250
200
150
100
50
0
Cassandra
MySQL
Posibles trueques

7.4.3. Combinación de lectura y escritura

A continuación describiremos, como en los casos anteriores, los accesos a la base de datos que se efectúan para las operaciones que se han medido.

Añadir ruta restantes para posible trueque

En esta operación, cuando se obtienen las rutas que tiene establecidas del usuario, se buscan usuarios para posibles trueques, lo que hace que sea una operación costosa.

En Cassandra se hacen dos accesos: uno para obtener las rutas que ya tiene el usuario y otra para añadir las rutas que faltan. Ambos accesos se realizan a una Column Family.

En MySQL se necesitan dos accesos también pero el dato a almacenar no es la ruta de las ofertas que le interese, si no el identificador del nodo.

Modificar título de las ofertas

Para esta operación, Cassandra necesita seis accesos a la base de datos: uno para obtener los datos de la oferta y los cinco restantes para la actualización de la oferta. El primero se realiza sobre una Column Family y, de los cinco restantes, cuatro se realizan a una Column Family y uno a una Super Column Family.

En cambio, MySQL necesita cuatro accesos: uno para obtener los datos de la oferta y tres más para actualizar la oferta.

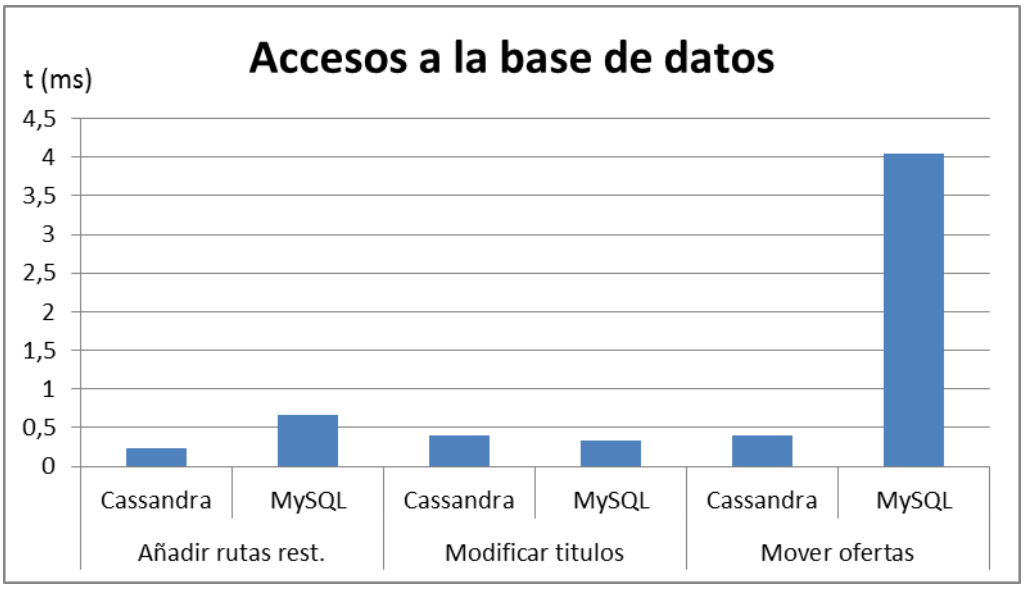
Mover ofertas de activas a inactivas

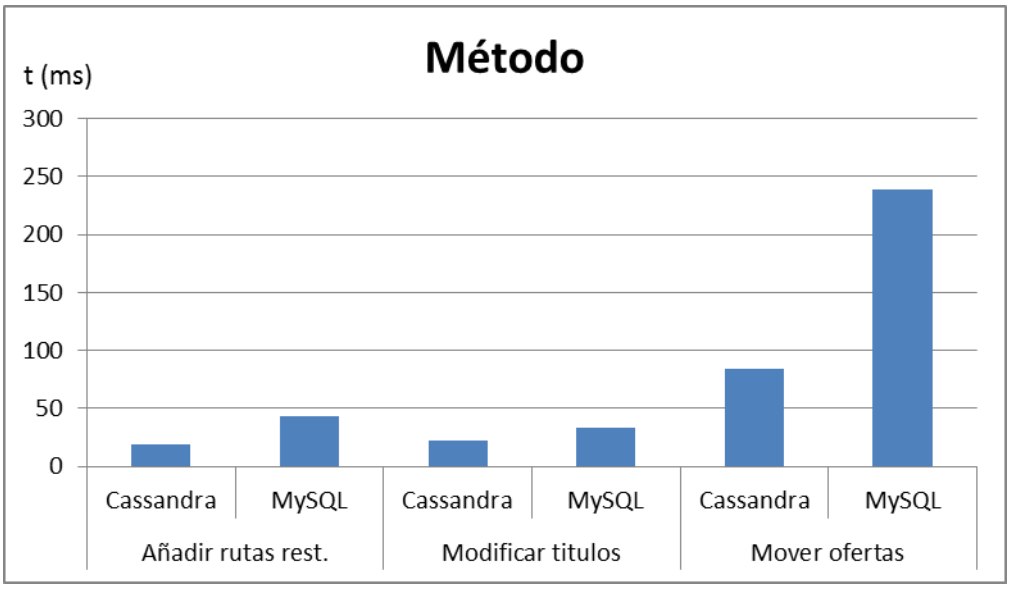
Para mover ofertas, Cassandra hace tres accesos a la base de datos: uno para obtener los identificadores de las ofertas y los otros dos para actualizar la oferta moviendo los identificadores de un estado a otro y actualizando la oferta. De estos accesos, dos son a Super Column Family y el otro es a una Column Family.

MySQL necesita dos accesos: uno para obtener los identificadores y otro para modificar las ofertas.

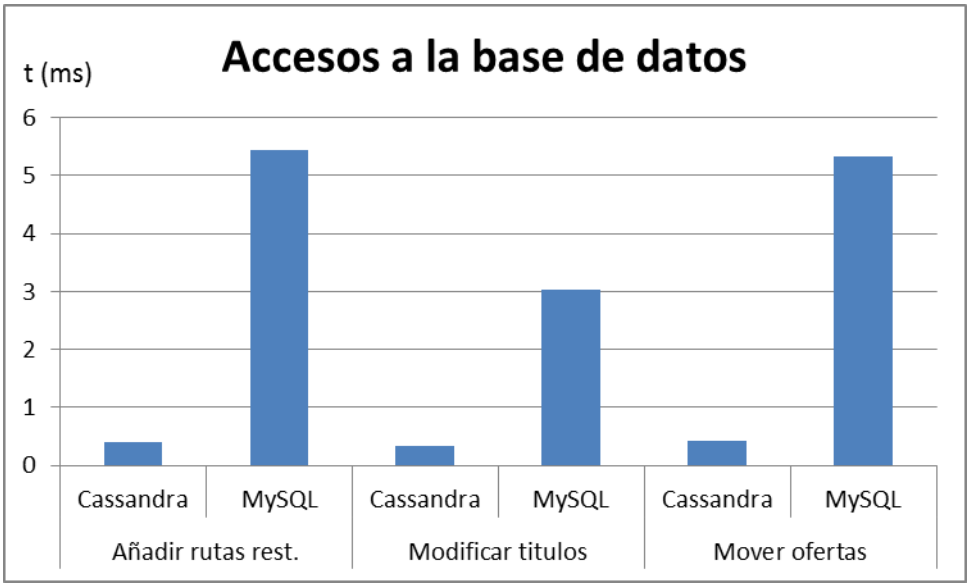
Gráficas

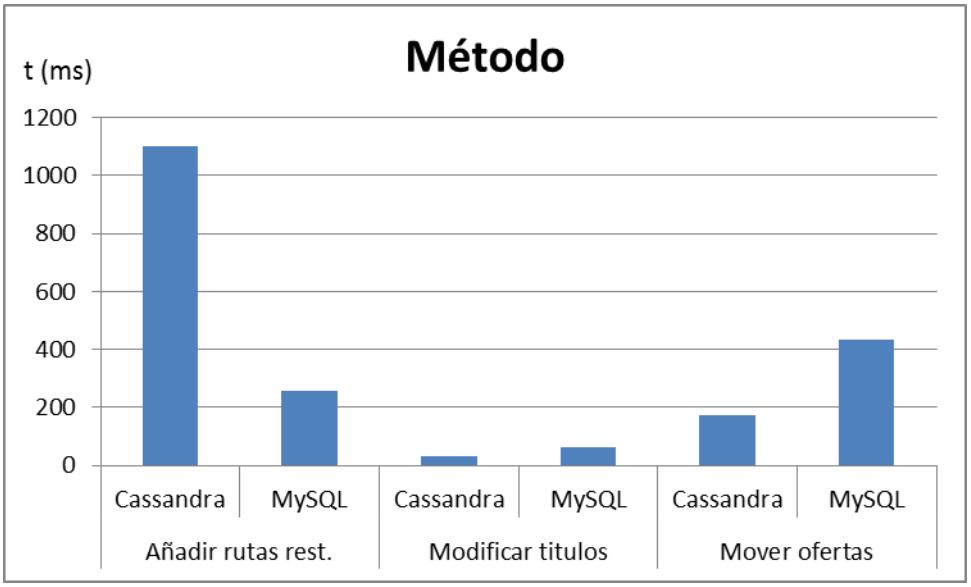
Estado 1



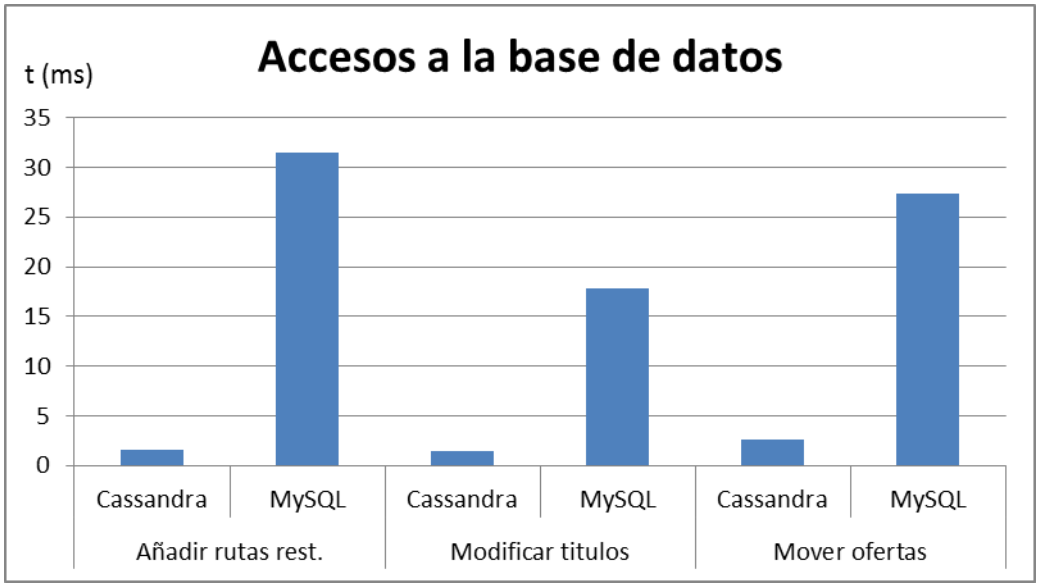


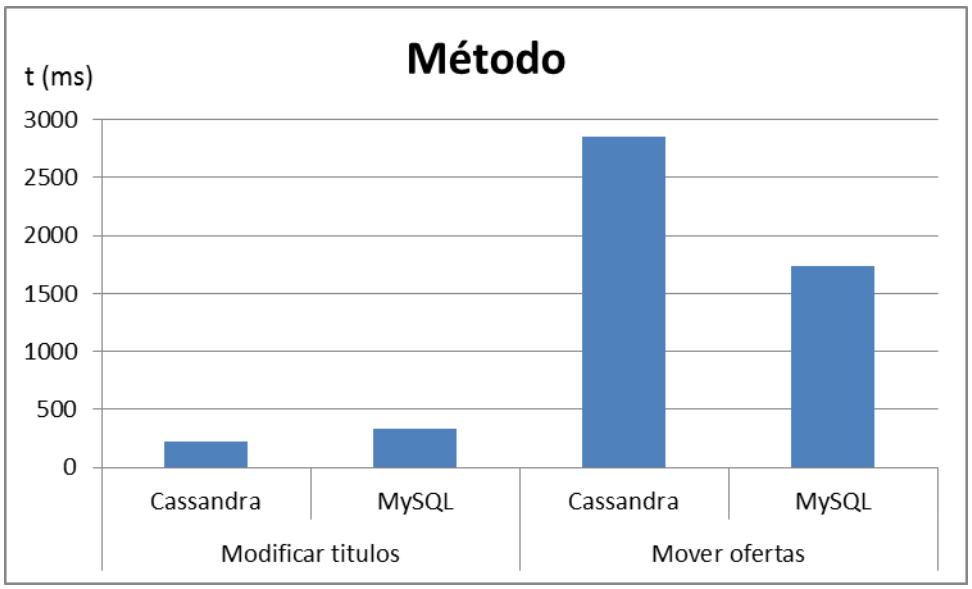
Estado 2

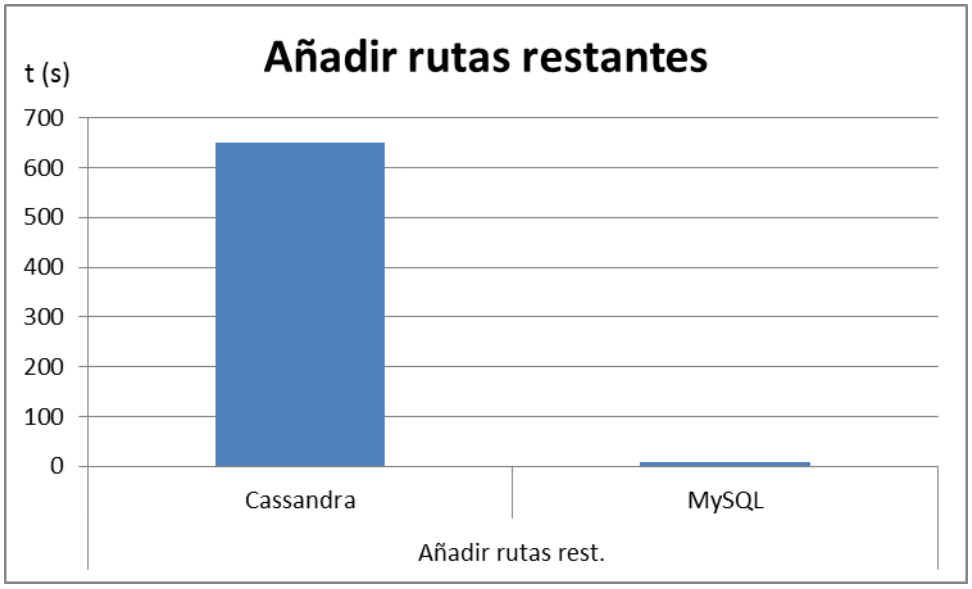

Accesos a la base de datos
t (ms)
6
5
4
3
2
1
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Añadir rutas rest.
Modificar titulos
Mover ofertas


Método
t (ms)
1200
1000
800
600
400
200
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Añadir rutas rest.
Modificar titulos
Mover ofertas

Estado 3


Accesos a la base de datos
t (ms)
35
30
25
20
15
10
5
0
Cassandra
MySQL
Cassandra
MySQL
Cassandra
MySQL
Añadir rutas rest.
Modificar titulos
Mover ofertas


Método
t (ms)
3000
2500
2000
1500
1000
500
0
Cassandra
MySQL
Cassandra
MySQL
Modificar titulos
Mover ofertas


Añadir rutas restantes
t (s)
700
600
500
400
300
200
100
0
Cassandra
MySQL
Añadir rutas rest.

7.5. Conclusiones

Podemos ver como Cassandra, en prácticamente todos los casos, es más rápido que MySQL ofreciendo los datos pedidos al usuario (gráficas Método). Esto es debido a que en MySQL se ha hecho uso de un Framework para la conversión de los objetos de las clases a objetos de la bases de datos, es decir, el mapeo objeto-relacional.

Si nos fijamos en los resultados de los accesos a las bases de datos, podemos ver como MySQL ofrece tiempos superiores cuando el tamaño de ésta es mayor. Éste es el problema que se mencionaba en la introducción, la indexación de tablas muy grandes.

Aunque Cassandra no esté optimizado para las operaciones de lectura, la diferencia entre ambos sistemas no es muy grande.

En el caso de añadir las rutas restantes, podemos observar como el tiempo medio del método en ofrecer los datos al usuario es muchísimo mayor en el caso de Cassandra. Este tiempo tan alto es debido al acceso que se realiza para obtener los posibles trueques de cada usuario que requiere una función de procesado de los datos para poder ser visualizados en el árbol de clasificación. El acceso a la base de datos del mismo método es tan inferior respecto al tiempo total porque se ha calculado la media de todos los accesos, lo que incluye: un conjunto de lecturas con tiempos muy altos y una única escritura para cada ruta con un tiempo medio de 1 milisegundo.

Antes de concluir con la evaluación de rendimiento, me gustaría remarcar dos cosas:

1. El modelo de datos que he diseñado no ha sido la mejor opción. Por este motivo, creo que el rendimiento real de Cassandra podría ser algo mayor.
2. Debemos pensar que Cassandra está diseñada para sistemas distribuidos. En el caso de un solo nodo no podemos aprovechar todo su potencial.

Conclusión: Cassandra ofrece tiempos mejores en prácticamente todos los casos. Además, Cassandra no necesita frameworks que hagan la conversión a objeto de la base de datos ya que, gracias a la API utilizada, los objetos son ya objetos de Cassandra.

8. CONCLUSIONES

Una vez implementada la aplicación FUFISSANDRA y ejecutadas las pruebas de rendimiento, he encontrado posibles mejoras en la aplicación.

La más destacable, por su diferencia con el modelo relacional, es la repetición de datos para reducir el número de accesos a la base de datos. Como he comentado anteriormente, la "mala" costumbre de relacionar los datos implica que el diseño del nuevo modelo de datos no se realice correctamente.

Otra mejora que se podría hacer es que gracias a la API utilizada, se podrían empaquetar todos los accesos a la base de datos que estén relacionados, como por ejemplo todos los datos relacionados con las personas o con las ofertas. Así, nos aseguraríamos de que si falla alguno de los accesos, la misma API realiza el "rollback" sin tener que realizarlo manualmente eliminando los datos que se hayan guardado.

Para realizar un proyecto como éste, ha sido necesario ampliar los conocimientos adquiridos a lo largo de la carrera de la Ingeniería Técnica en Informática de Sistemas.

Dado que esta ingeniería está más orientada a nivel hardware que a nivel de software, mis conocimientos de ingeniería del software son insuficientes. Un diagrama de clases o la organización de la arquitectura de la aplicación por capas son conceptos que no había escuchado antes.

Cambiar del modelo relacional al no relacional no es tarea sencilla. Es inevitable hacer comparaciones entre ambos (yo mismo las he hecho describiendo el modelo de datos de Cassandra), pero para diseñar un modelo de datos no relacional correctamente, deberíamos olvidarnos de los conceptos del modelo relacional.

Estaba planificado ejecutar pruebas de rendimiento en un clúster ya que es el escenario habitual de las aplicaciones web escalables, pero debido a incidencias técnicas no se han podido realizar. Personalmente, me hubiera gustado ejecutar las dos aplicaciones en un sistema como este porque plantea un nuevo reto a los conocimientos de ingeniero adquiridos en la carrera.

Realizar un proyecto como éste no es nada fácil. Han sido muchas horas de estudio de tecnologías que desconocía, muchas horas analizando una aplicación ya desarrollada, muchas horas implementando y probando las funcionalidades de esta aplicación y, sobre todo, muchas horas ejecutando las pruebas de rendimiento. Aunque ha sido un camino muy duro, la experiencia ha sido muy gratificante. Teniendo en cuenta que la idea inicial del proyecto era un cambio de la base de datos de FUFIFU y la versión móvil de la web, modificar el proyecto me ha aportado conocimientos que, tal vez, no hubiera adquirido.

9. REFERENCIAS Y BIBLIOGRAFÍA

1. Cassandra: The Definitive Guide
   Eben Hewitt, Editorial O'Reilly. Noviembre de 2010.
   http://javabloger-mini-books.googlecode.com/files/Oreilly.Cassandra.The.Definitive.Guide.Nov.2010.pdf

2. Bases de dades basades en columnes
   Emili Calonge Sotomayor, director David Carrera Pérez. Enero de 2010
   http://upcommons.upc.edu/pfc/handle/2099.1/11406

3. FUFIFU – Xarxa d'intercanvis sense diners
   Luis Barguñó Jané, director Manel Guerrero Zapata. Enero de 2010

4. Documentación de Apache Cassandra 1
   Apache Foundation.
   http://wiki.apache.org/cassandra/

5. Documentación de Apache Cassandra 2
   DataStax web page.
   http://www.datastax.com/docs/0.7/index

6. Clientes para la comunicación con Cassandra
   Apache Foundation.
   http://wiki.apache.org/cassandra/ClientOptions07

7. Documentación de HECTOR
   Varios Autores.
   https://github.com/rantav/hector
   https://github.com/rantav/hector/wiki/User-Guide

8. Documentación de PELOPS
   Varios Autores.
   http://ria101.wordpress.com/2010/06/11/pelops-the-beautiful-cassandra-database-client-for-java/

9. Documentación de KUNDERA
   Varios Autores.
   https://github.com/impetus-opensource/Kundera/

10. Documentación de DEMOISELLE
    Varios Autores.
    http://www.frameworkdemoiselle.gov.br/ultimas-noticias/projetos
    http://pt.wikipedia.org/wiki/Demoiselle_Framework
    http://demoiselle.sourceforge.net/component/demoiselle-cassandra/1.0.0/

11. Documentación de PYCASSA
Varios Autores.
http://en.wikipedia.org/wiki/Pycassa
https://github.com/pycassa/pycassa
http://pycassa.github.com/pycassa/

12. Documentación de TELEPHUS
Varios Autores.
https://github.com/driftx/Telephus

13. Documentación de DJANGO CASSANDRA BACKEND
Varios Autores.
https://github.com/vaterlaus/django_cassandra_backend

14. Documentación de PHPCASSA
Varios Autores.
https://github.com/thobbs/phpcassa

15. Documentación SimpleCassie
Varios Autores
http://code.google.com/p/simpletools-php/wiki/SimpleCassie

16. Documentación de AQUILES
Varios Autores
http://aquiles.codeplex.com/

17. Documentación de FLUENT
Varios Autores.
https://github.com/managedfusion/fluentcassandra

18. Documentación de FAUNA
Varios Autores.
http://fauna.org/fauna/cassandra/files/README_rdoc.html

19. Documentación de THRIFT
Varios Autores.
http://en.wikipedia.org/wiki/Apache_Thrift
http://en.wikipedia.org/wiki/Interface_definition_language
http://en.wikipedia.org/wiki/Remote_procedure_call

20. Nombres y apellidos de Catalunya 2010
IDESCAT, Institut D'EStadística de CATalunya. Enero del 2011
http://www.idescat.cat/noms/?lang=es
http://www.idescat.cat/cognoms/?lang=es

21. Generación de puntos aleatorios
GeoMidpoint.
http://www.geomidpoint.com/random/
http://www.geomidpoint.com/random/calculation.html