FACULTAD DE CIENCIAS

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por:	Ignacio Sañudo Olmedo
Director del PFC:	Pablo Sánchez Barreiro
Título:	Desarrollo de un Entorno Dirigido por Modelos para la Creación de Esquemas de Bases de Datos en Cassandra a partir de Modelos UML
Title:	Development of a Model-Driven Environment for Creating Database Schemas in Cassandra from UML Models

Presentado a examen el día:

para acceder al Título de
INGENIERO EN INFORMÁTICA

Composición del Tribunal:

Presidente (Apellidos, Nombre):
Secretario (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):
Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: Vocal

Fdo.: Vocal

Fdo.: Vocal

Fdo.: El Director del PFC

II

Agradecimientos

A mi familia y amigos por haberme apoyado durante estos cinco años, en especial a mis padres que me han permitido estudiar la carrera que he querido y me han apoyado en todo momento.

A mi director Pablo Sánchez por darme la oportunidad de realizar este proyecto y guiarme durante su desarrollo.

A todos mis compañeros de clase por los momentos que hemos vivido dentro y fuera de las aulas.

A todos los profesores y trabajadores de la Universidad de Cantabria que se han preocupado en proporcionarnos un servicio de calidad.

Resumen

Ciertas aplicaciones de alta demanda, accesibles a través de internet, como Twitter o Amazon, poseen requisitos muy particulares que resultan complejos de satisfacer utilizando los tradicionales sistemas gestores de bases de datos relacionales. Para resolver este problema, han ido apareciendo en los últimos años una serie de tecnologías de almacenamiento y recuperación de datos conocidas como NoSQL. No obstante, dichas tecnologías han aparecido a nivel de implementación, no siendo posible aún construir sistemas no relacionales desde modelos conceptuales de alto nivel, tal como se ha venido realizando para el caso relacional desde hace décadas.

El objetivo de este proyecto es crear haciendo uso de las modernas tecnologías de desarrollo software dirigido por modelos, una herramienta que permita transformar un modelo de datos conceptual de alto nivel expresado en UML 2.0 en una implementación para un sistema de almacenamiento de datos NoSQL. Concretamente se utilizará el sistema de datos basado en columnas llamado Cassandra.

Palabras Clave

Desarrollo Dirigido por Modelos, Ingeniería Dirigida por Modelos, Generación de Código, Cassandra, Epsilon, UML, CQL.

VI

Preface

Certain high demand applications accessible via internet like Twitter or Amazon, have very specific requirements that are complex to meet using traditional relational databases. To resolve this problem have appeared in the last few years technologies of data storage known as NoSQL. However, these technologies have appeared at the implementation level, still not possible to build non-relational systems from high-level conceptual models, as has been done for relational systems for decades.

The objective of this project is create a tool using modern techniques known as Model-Driven Development, this tool must transform a conceptual data model expressed in high-level (UML 2.0) in a data repository for a NoSQL storage system data. Specifically, we will use the data system based on columns called Cassandra.

Keywords

Model-Driven Development,Model-Driven Engineering, Code Generation, Cassandra, Epsilon, UML, CQL.

VIII

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este capítulo sirve de introducción a la presente Memoria de Proyecto de Fin de Carrera. En primer lugar se realiza una breve introducción de los conceptos generales del proyecto. La siguiente sección describe conceptos orientados a la Ingeniería de Lenguajes Dirigida por Modelos y al Desarrollo Dirigido por Modelos, términos clave para entender el contexto del proyecto. La tercera sección expone la planificación que ha seguido el proyecto para su realización, desde formación hasta etapas de desarrollo. La última sección está dedicada a describir la estructura del documento.

Índice

1.1. Contexto del proyecto

La irrupción de internet en diferentes ámbitos de nuestra vida diaria ha dado origen a diversas aplicaciones, tales como *Amazon*, *iCloud*, *Skype* o *Twitter*, las cuales, poco a poco se han ido incorporando de una manera u otra a nuestra rutina diaria. Hoy en día, por ejemplo, se mide el impacto de un evento a partir de su repercusión en Twitter, las familias dispersas geográficamente se comunican a través de Skype y como indicador del éxito de una nueva canción se usa su número de descargas desde Amazon.

Dichas aplicaciones, por sus peculiaridades propias y por el hecho de estar disponibles globalmente a través de internet, presentan ciertos requisitos, normalmente no funcionales, que son, en cierto modo, muy diferentes de los requisitos que presentaban las aplicaciones que se habían venido desarrollando hasta ahora.

Por ejemplo, la disponibilidad (*availability*, en inglés) se está convirtiendo en un aspecto critico en estos sistemas, ya que una caída del sistema puede generar pérdidas multimillonarias. Por ejemplo, las pérdidas por interrupción del servicio de Amazon se estiman en más de 60.000$ Clay (2013). En otros casos, como por ejemplo Twitter, la aplicación debe soportar una alta carga de usuarios concurrentes, donde estos usuarios realizan acciones muy simples y sencillas, que apenas precisan de soporte transaccional.

Los sistemas gestores de bases de datos relacionales suelen presentar dificultades a la hora de satisfacer esta clase de nuevos requisitos. Por tanto, con el objetivo de adecuarse a estas nuevas aplicaciones, han ido apareciendo en los últimos años una serie de nuevas tecnologías para la gestión y almacenamiento de datos que se han ido denominando comúnmente como NoSQL (*Not Only SQL*) Stonebraker (2010).

Las tecnologías NoSQL sacrifican algunas de las ventajas bien conocidas de los sistemas de gestión de bases de datos relacionales, como la integridad de datos o el soporte transaccional, con el fin de proporcionar una mejor escalabilidad y un mayor rendimiento. Siguiendo esta idea, varios sistemas NoSQL, como *Cassandra* Lakshman (2010), HBase George (2011) o MongoDB Plugge (2010), han aparecido en los últimos años.

Estas tecnologías han ido apareciendo hasta ahora en el nivel de aplicación. Es decir, actualmente se dispone de sistemas gestores de datos que dan soporte a estas nuevas tecnologías NoSQL. Sin embargo, las tecnologías NoSQL no están aún adecuadamente integradas en procesos de desarrollo software. Es decir, los ingenieros software no disponen de procesos bien definidos para generar una implementación NoSQL de un modelo de datos de alto nivel.

Cabe destacar que, por contra, estos procesos bien definidos sí existen en el caso de los sistemas relacionales, donde han estado disponible desde hace décadas. El ejemplo más conocido sea quizás el proceso de transformación de modelos entidad-relación en implementaciones relaciones Elmasri and Navathe (2010).

Dentro del Departamento de Ingeniería Informática y Electrónica de la Universidad de Cantabria, varios profesores han definido una serie de reglas para transformar modelos conceptuales de datos, expresados en UML 2.0 Olivé (2007), en implementaciones para Cassandra Lakshman (2010), un sistema gestor de datos NoSQL basado en columnas.

El objetivo de este presente Proyecto Fin de Carrera es el de construir una herramienta o generador de código que permita automatizar dichas reglas de transformación, utilizando para ello técnicas de desarrollo software dirigido por modelos Marco Brambilla (2012). Las siguientes secciones describen una serie de conceptos sobre las técnicas de desarrollo software dirigido por modelos que son necesarios para comprender la metodología de trabajo seguida en este proyecto, la cual se describe al final de este capítulo.

1.2. Antecedentes

Según Cáceres (2008), la *Ingeniería Dirigida por Modelos* o *Model-Driven Engineering* (MDE) puede ser definida como la *"técnica que hace uso, de forma sistemática y reiterada, de modelos como elementos primordiales a largo de todo el proceso de desarrollo software."*. En un proceso de desarrollo software dirigido por modelos, se crean modelos de alto nivel del sistema a desarrollar, los cuales se van refinando a través de distintas etapas, hasta obtener una implementación completa del mismo. Dicho proceso de refinamiento se intenta automatizar, en la medida de lo posible, por medio de transformaciones entre modelos, o modelo a texto. El *Desarrollo Dirigido por Modelos* (MDD) se puede definir como un enfoque de la Ingeniería del software y de la Ingeniería Dirigida por Modelos (MDE) que utiliza el modelo para crear un producto. ¿Y que es un modelo?. Un modelo se puede entender como la descripción o representación de un sistema en un lenguaje bien definido. Para entender lo que representa un modelo dentro del Desarrollo Dirigido por Modelos hay que saber previamente lo que es un metamodelo. Un metamodelo es un modelo usado para especificar un lenguaje, básicamente describe las características del lenguaje. Por lo tanto un modelo se puede entender como la instancia de un metamodelo. El resultado de la utilización del desarrollo dirigido por modelos es traducido en reducción de costes debido a que el recurso humano requerido es menor, un aumento de la productividad y reutilización de componentes, además se puede aumentar el nivel de abstracción a la hora de realizar el diseño de un software. Para poder manipular los modelos de un proceso de desarrollo, necesitamos que dichos modelos sean procesables por una computadora. Esto se consigue definiendo formalmente lenguajes de modelado mediante una estructura compuesta de un metamodelo al menos, más una sintaxis concreta y una semántica. A continuación, describimos cada uno de estos elementos.

Un lenguaje de modelado está formado de sintaxis y semántica. La sintaxis se puede entender como el conjunto de normas o reglas de escritura que ha de tener el lenguaje. La semántica refleja el significado de la sintaxis. Un metamodelo es la representación o modelo de la sintaxis del lenguaje que se desea implementar Kleppe (2009). Por tanto, cuando tratamos de definir un nuevo lenguaje de modelado, lo primero que debemos hacer es construir su metamodelo. Para ello se utilizan lenguajes de metamodelo, como Ecore Budinsky (2009) o MOF OMG (2007), los cuales permiten especificar mediante diagramas de clases qué elementos contiene el lenguaje, qué atributos tienen esos elementos y cómo se relacionan.

La Figura 1.1 muestra la representación de la sintaxis abstracta o metamodelo del lenguaje de entrada. El metamodelo consiste en un sencillo grafo el cual está compuesto de un número indefinido de nodos, estos nodos están conectados entre sí mediante aristas. Estos nodos, como se ve en la imagen, tienen un atributo de tipo Tcolor por lo que un Nodo puede ser de color rojo

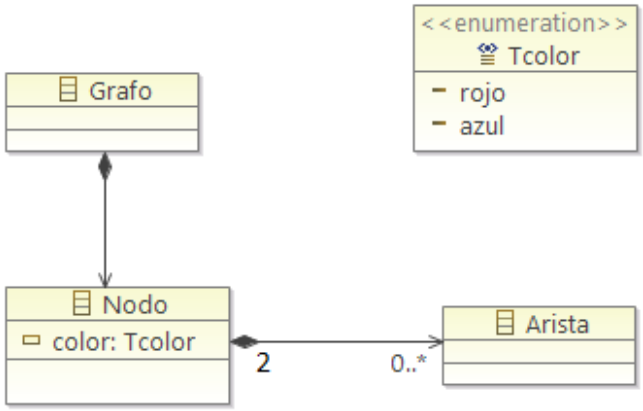

Figura 1.1: Metamodelo del grafo

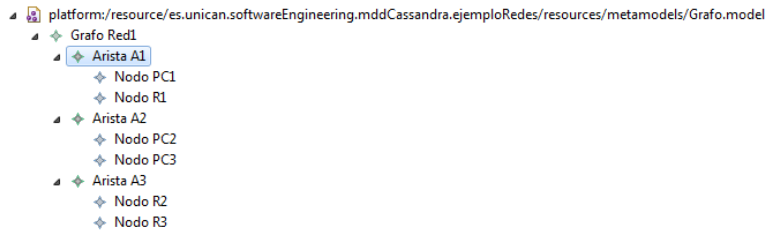

Figura 1.2: Sintaxis concreta del grafo

o de color azul.

El siguiente paso en la definición de un lenguaje consiste en la definición de su *sintaxis concreta*. En este paso debemos básicamente especificar cómo se representa cada uno de los elementos definidos en el metamodelo, en la Figura 1.2 podemos ver una instanciación ejemplo del metamodelo Grafo. Como podemos ver es la definición de cada uno de los elementos del metamodelo, el modelo cuenta con tres Aristas y por cada Arista dos Nodo conectados entre si, cada Nodo con un color y nombre determinado.

Por último, la semántica describiría el significado de cada elemento, o cómo debe interpretarse cada elemento de un modelo. Por ejemplo, en un modelo de grafos, una arista entre dos nodos puede significar, si así lo especificamos en la semántica, que los nodos conectados por la arista pueden comunicarse entre ellos mediante algún mecanismo no definido.

Una vez que tenemos bien definidos una serie de lenguajes de modelado, el siguiente elemento que nos haría falta para poder tener un proceso de desarrollo software dirigido por modelos serían transformaciones automáticas entre modelos. La idea es que las transformaciones se puedan describir de forma algorítmica e implementar en algún lenguaje, de forma que la transfor-

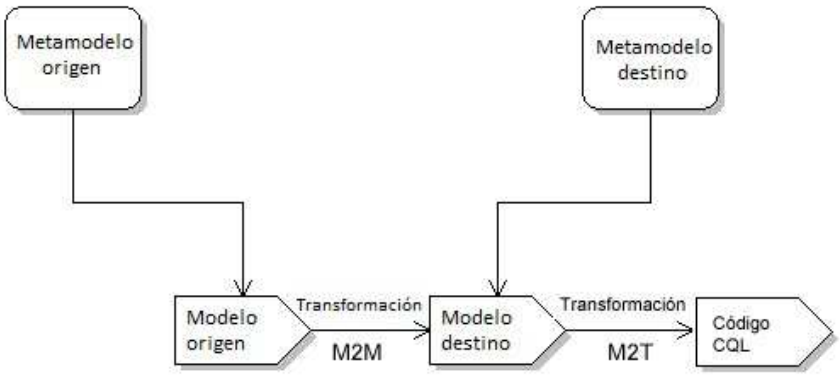

Figura 1.3: Proceso de transformación de un modelo

mación acepte como entrada un modelo y genere como salida el refinamiento o transformación de ese modelo.

Para implementar estas transformaciones, se usan una serie de lenguajes específicos para tal propósito, los cuales pueden estructurarse en dos grupos claramente diferenciados.

1. *Model to model (M2M).* Dado un modelo de entrada esta transformación produce otro modelo como salida. El modelo de salida puede ser un modelo en el mismo lenguaje que el original de entrada o un modelo en un lenguaje distinto. En el primer caso, el modelo de salida suele ser una variación del modelo de entrada, al cual se le añaden nuevos elementos y se modifican algunos de los existentes.

2. *Model to text (M2T).* Dado un modelo de entrada, genera texto como salida. Este texto puede ser código de un lenguaje de programación, texto plano para un manual de usuario, etc.

Para explicar todo este proceso se presenta un sencillo ejemplo donde se explican cómo se utilizan las técnicas de desarrollo dirigido por modelos, Model to Model (M2M) y Model to Text (M2T). Para aplicar M2M nos harán falta dos metamodelos (uno origen y uno destino) y para aplicar M2T utilizaremos como destino código HTML. Podemos ver una imagen del proceso llevado a cabo en la Figura 1.3.

Para presentar este ejemplo utilizaremos el metamodelo de la Figura 1.1 y a continuación se presenta otro metamodelo (Figura 1.4). Este último metamodelo consiste en la representación de una red de computadores que está compuesta de PC's y Routers, conectados indistintamente entre sí mediante cables. Tanto los nodos como los cables tiene un atributo de tipo String para definir su nombre.

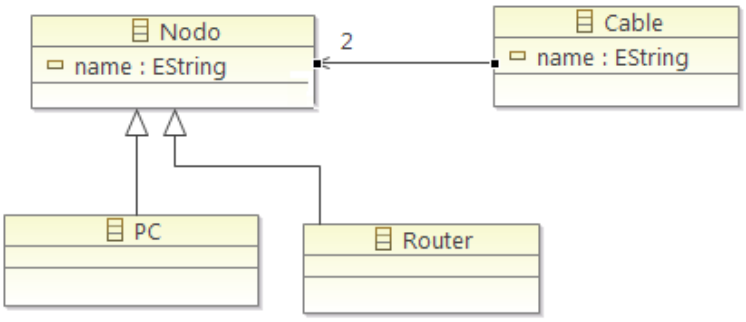

Figura 1.4: Metamodelo de la red

Nuestro objetivo es la transformación de un modelo de tipo Grafo a un modelo de tipo Red, dependiendo del color del Nodo, el Nodo será transformado en un PC o en un Router (Rojo-PC, Azul-Router). Para realizar esta transformación se utilizan técnicas M2M utilizando para ello el lenguaje *Epsilon Transformation Language* (ETL).

En primer lugar necesitamos definir el modelo del grafo para poder realizar la transformación a un modelo de tipo Red. Este modelo es una instancia del metamodelo. La figura 1.5 muestra el código que realiza el proceso de transformación de un Grafo a una Red.

En este código encontramos solo una regla. Esta regla transforma aristas del grafo a cables de la red. La primera instrucción copia el nombre de la arista al cable. A continuación la primera condicional cuestiona si esa arista tiene un padre definido en caso afirmativo asigna el cable a la red. A continuación por cada nodo se crea o bien un PC o un Router dependiendo del color del nodo que se esté analizando (rojo-PC, azul-Router). En siguiente lugar se asigna el nodo creado al cable correspondiente y finalmente se añade a la red. Este proceso se repite por cada arista del grafo. Una vez ejecutado este código dado un modelo de entrada de tipo Grafo obtenemos un modelo equivalente de tipo Red que cumple las reglas definidas en el meta-modelo.

Una vez obtenido el modelo de tipo Red, podemos crear una representación textual del modelo generado en HTML utilizando técnicas M2T. La Figura 1.6 muestra como se realiza la generación de código HTML utilizando para ello el modelo generado de una red a partir de un grafo, para realizar esta transformación se utiliza el lenguaje *EGL* (Epsilon Generation Language).

La representación en HTML del código generado es el que se puede observar en la Figura 1.7.

La siguiente sección describe la metodología y objetivos parciales que habrá de ir satisfaciendo este proyecto.

```
Grafo2Red.etl
-------------------------------------------------------
01 rule Arista2Cable
02 transform a : Grafo!Arista
03 to r : Red!Cable {
04     r.nameCable = a.nombreArista;
05     if (a.parent.isDefined()) {
06         r.parent=Red;
07         for (nodoArista in a.children) {
08            if(nodoArista.color=TColor#R){
09                 var PC : new Red!PC;
10                 PC.nameNodo="PC"+iPC;
11                 r.children.add(PC);
12                 iPC=iPC+1;
13             }
14            else{
15                 var Router : new Red!Router;
16                 Router.nameNodo="Router"+iRouter;
17                 r.children.add(Router);
18                 iRouter=iRouter+1;
19             }
20         }
21     }
22 }
-------------------------------------------------------
```

Figura 1.5: Ejemplo código ETL

1.3. Planificación del proyecto

El objetivo de este proyecto de fin de carrera es la implementación de un generador de código Cassandra a partir de modelos UML. El proceso de desarrollo así como el de aprendizaje que se ha seguido para la realización del proyecto es descrito a continuación. En la Figura 1.8 se puede observar la planificación que se ha llevado a cabo.

La primera tarea como se ve en la Figura 1.8, *T1A* consistió en la adquisición de los conocimientos teóricos que fueron necesarios para la realización del proyecto. Dentro de esta fase de aprendizaje hay tres conceptos pilares para la realización del proyecto, el primer area de estudio fue el *Desarrollo Dirigido por Modelos*, este concepto es clave para entender en que consiste la metodología para la construcción del generador de código. A continuación la siguiente materia clave consiste en la creación de lenguajes de modelado Kleppe (2009) esto nos ayuda de cara a la construcción y manipulación de metamodelos. Finalmente el estudio de *Cassandra*, funcionamiento de la arquitectura y lenguaje de consultas, de esta manera podemos hacer una implementación del generador de código que sea completamente funcional.

La siguiente tarea también es de adquisición de conceptos pero a nivel práctico como se puede ver en la Figura 1.8, *T1B*. Diferenciamos dos

```
Red.egl
-------------------------------------------------------
01 [%
02     var red: Red := Red.allInstances().at(0);
03 %]
04
05 <html>
06     <head>
07         <title> Red </title>
08     </head>
09     <body>
10         <h1>Conexiones</h1>
11         <table  border="1">
12             <col style="width: 200px" />
13             <col style="width: 100px" span="3" />
14             [% for (conexiones in red.conexiones){%]
15             <tr>
16                 <th scope="row">[%=conexiones.nameCable%]</th>
17                 [% for (nodos in conexiones.children){%]
18                     <td>[%=nodos.nameNodo%]</td>
19                 [% }%]
20             </tr>
21             [% }%]
22         </table>
23     </body>
24 </html>
-------------------------------------------------------
```

Figura 1.6: Ejemplo código EGL

Conexiones

A1	PC1	Router1
A2	PC2	PC3
A3	Router2	PC4

Figura 1.7: Resultado transformación HTML

fases practicas: En primer lugar la toma de contacto con la herramienta *Epsilon* Dimitris Kolovos (2014) y los lenguajes de modelado, EOL (*Epsilon Object Language*), ETL (*Epsilon Transformation Language*) y EGL (*Epsilon Generation Language*) a base de ejemplos prácticos. Estos ejemplos han sido descritos en la sección anterior. Así como la practica con EMF (*Epsilon Modeling Framework*) Dave Steinberg (2008) el lenguaje para la definición de lenguajes de modelado. En segundo lugar fue necesaria práctica con la shell de Cassandra para probar la sintaxis del lenguaje de consultas CQL

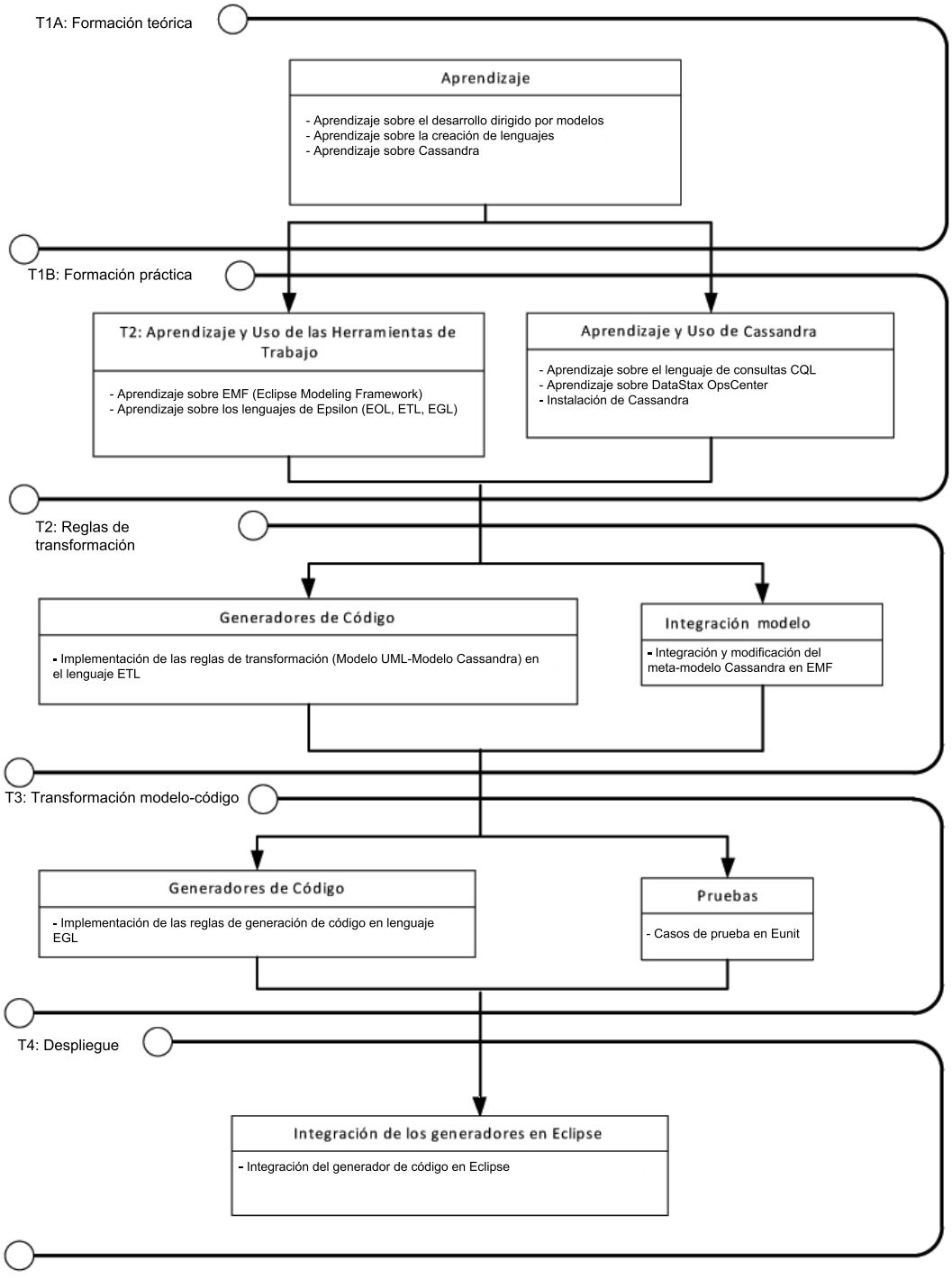

Figura 1.8: Proceso de desarrollo del Proyecto Fin de Carrera

Cassandra Query Language. Una vez conocido estos conceptos se estudiaron

las reglas de transformación a aplicar para transformar un modelo UML a un modelo Cassandra, estas reglas fueron propuestas por Pablo Sánchez (2013).

Las tareas *T3* y *T4* consisten en el desarrollo del generador de código. Para ello, previa implementación habrá que transformar el modelo UML a un modelo escrito en Cassandra, para lograr este objetivo será necesario definir una serie de reglas de equivalencias entre modelos utilizando para ello el lenguaje ETL. Por lo tanto necesitaremos desarrollar un metamodelo que describa el modelado de Cassandra y un metamodelo que describa el modelado de UML (este último metamodelo lo proporciona la herramienta con la que vamos a trabajar). La herramienta utilizada para modelar el metamodelo es EMF. Estas tareas corresponden a *T3* de la Figura 1.8. La tarea *T4* corresponde a la transformación de dicho modelo Cassandra a código CQL, para lograr esta transformación utilizamos el lenguaje EGL. Una vez terminado se comenzaron las pruebas utilizando la herramienta EUnit.

Por último la fase de despliegue (Figura 1.8, *T4*) consiste en la integración del generador de código en la herramienta Eclipse con un plugin en el que se introduce el modelo UML y se genera el código correspondiente. Sin embargo por problemas de compatibilidad de la herramienta Epsilon con Eclipse no ha sido posible realizar con éxito esta fase de despliegue en su totalidad. Este error ha sido reportado a los desarrolladores de Epsilon.

Como resultado del proyecto se han implementado las reglas de transformación modelo-modelo y modelo-código de manera que, podemos transformar modelos UML 2.0 en código para la creación de un repositorio de datos en Cassandra escrito en el lenguaje de consultas de Cassandra CQL que puede ser ejecutado en cualquier herramienta que soporte dicho lenguaje.

1.4. Estructura del Documento

Tras este capítulo de introducción, la presente memoria del Proyecto Fin de Carrera se estructura tal y como se describe a continuación: El Capitulo 2 describe en términos generales todos los conceptos y tecnologías que son necesarios para comprender el contenido de esta memoria conceptos tales como Cassandra o EMF por ejemplo. El Capitulo 3 describe el proceso de desarrollo y creación de las reglas para realizar la transformación entre modelos. El Capítulo 4 explica como se ha realizado el generador de código así como las técnicas y tecnologías utilizadas. Finalmente el Capítulo 5 presenta mis conclusiones tras la realización del proyecto así como posibles mejoras de la herramienta, todo como cierre de la memoria del proyecto de fin de carrera.

Capítulo 2

Antecedentes

Este capítulo describe las tecnologías y técnicas utilizadas en el desarrollo del presente Proyecto de Fin de Carrera. La primera sección está dedicada a introducir el caso de estudio que se analizará a lo largo del presente proyecto. La siguiente sección está dedicada a describir en qué consiste Eclipse Modeling Framework (EMF) la herramienta utilizada para desarrollar lenguajes de modelado. A continuación se describe la herramienta Epsilon, herramienta que se ha utilizado para desarrollar el generador de código. La siguiente sección está dedicada a explicar de manera breve algunos conceptos de Cassandra.

Índice

2.1. Caso de estudio: Generador de código CQL-Cassandra

Esta sección presenta *Twissandra*, el caso de estudio que se utilizará lo largo de este proyecto. Twissandra es un proyecto creado para aprender como utilizar Cassandra. El modelo UML correspondiente a Twissandra se puede ver en la figura 2.1. Twissandra es una versión simplificada de Twitter.

Twitter[1] es una red social de microblogging, actualmente está muy extendida, que permite escribir a sus usuarios pequeños mensajes de texto, denominados *tweets*. Un *tweet* es simplemente un texto con un límite de 140 caracteres publicado a una hora determinada. La colección de todos los

[1] https://twitter.com/

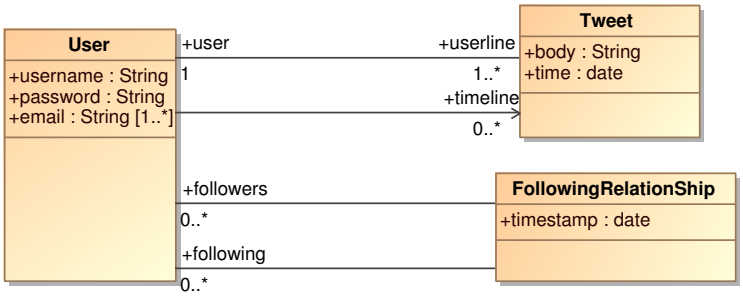

Figura 2.1: Modelo UML Twissandra

tweets publicados por un usuario cronológicamente ordenados determinan su *userline*.

Cada usuario registrado en Twitter puede seguir a otros usuarios registrados. Cuando decimos, por ejemplo, que Pedro sigue a María significa que Pedro recibirá en su cuenta todos los mensajes que publique María. En este caso, se dice que Pedro es un *follower* de María. De esta forma, cada usuario tiene asociado un *timeline* que no es más que la colección de todos los *tweets* publicados por las personas a las que sigue ordenados cronológicamente, es decir, por fecha de publicación.

Los principales casos de uso de *Twissandra* son:

1. Obtener el *timeline* de un usuario determinado.
2. Obtener el *userline* de un usuario determinado.
3. Obtener la lista de usuarios que un usuario está siguiendo.
4. Obtener la lista de usuarios que están siguiendo a un usuario específico.

Las dos últimas listas deben se deben devolver ordenadas cronológicamente.

Una vez entendido esto en los siguientes capítulos se procede a detallar el proceso de creación de un repositorio de datos en Cassandra que cubra los casos de usos citados anteriormente describiendo los procesos de transformación entre modelos que se realizaran así como la generación de código del repositorio de datos en Twissandra.

2.2. EMF

Una vez que hemos definido un metamodelo en Ecore, es posible crear instancias del mismo, aún no disponiendo de sintaxis concreta para el mismo. Además, EMF permite generar una serie de clases Java para la manipulación de dicho modelo, un esquema XML para poder persistir los modelos creados

y las facilidades necesarias para poder crear una instancia de dicho esquema para un modelo dado.

Con el paso de los años, EMF y Ecore se han convertido en los estándares de facto para la ingeniería software dirigida por modelos. De esta forma, cada nueva herramienta de modelado que ha ido surgiendo, no sólo es compatible con Ecore, sino que exige que los modelos que manipula posean un metamodelo definido en Ecore. Por ejemplo, prácticamente la totalidad de los lenguajes de transformación de modelos sólo aceptan como entrada y generan como salida modelos conformes a metamodelos definidos en Ecore. En adelante, nos referiremos a los modelos conformes a metamodelos definidos en Ecore simplemente como modelos Ecore, por simplicidad.

Dentro de este Proyecto Fin de Carrera, Ecore se ha utilizado para definir un metamodelo para Cassandra, que sirva como representación intermedia del código Cassandra y que facilite su posterior generación.

2.3. Epsilon

Epsilon Dimitris Kolovos (2014) es una familia de lenguajes y herramientas para el desarrollo de software dirigido por modelos. Entre las herramientas de esta familia encontramos lenguajes para realizar transformaciones modelo a modelo, transformaciones modelo a texto, herramientas para definir sintaxis concretas o para chequear la corrección sintaxis de un modelo, entre otras funcionalidades. Epsilon es distribuido a través de la plataforma de modelado de lenguajes de Eclipse.

De entre todas las herramientas proporcionadas por Epsilon, este Proyecto Fin de Carrera utiliza las siguientes:

***EOL* (Epsilon Object Language)** Es el lenguaje básico de Epsilon, utiizado por todos sus otros lenguajes, para la manipulación de objetos.

***ETL* (Epsilon Transformation Language)** Es el lenguaje de transformación modelo a modelo de Epsilon.

***EGL* (Epsilon Generation Language)** Es el lenguaje de transformación modelo a texto de Epsilon.

EUnit Es la herramienta proporcionada por Epsilon para la definición y ejecución de los casos de prueba que permiten verificar el correcto funcionamiento de las transformaciones desarrolladas.

A continuación describimos con más detalle cada uno de estos lenguajes.

2.3.1. Epsilon Object Language

Epsilon Object Language (EOL) es un lenguaje de programación imperativo utilizado para crear, consultar y modificar modelos Ecore. EOL se

```
-------------------------------------------------------
01  for (nodo in Arista.children){
02      if(nodo.color=TColor#B){
03          numeroNodos=numeroNodos+1;
04      }
05 }
06 //Podemos realizar lo mismo de la siguiente manera:
07
08 Nodo.all.select(r|r.TColor#B==true).size().println();
-------------------------------------------------------
```

Figura 2.2: Ejemplo código EOL

puede considerar un lenguaje mezcla de Javascript y OCL, que combina lo mejor de ambos lenguajes. Como tal, proporciona todas las características habituales imperativas que se encuentran en Javascript (por ejemplo, los bucles for y while) y todas las características interesantes de OCL como los filtros sobre colecciones, como por ejemplo Sequence{1..5}.select(x | x > 3).

Partiendo del metamodelo *Grafo* definido en el capítulo anterior (Figura 1.1) se presenta el siguiente ejemplo para entender como funciona EOL (Figura 2.2). En este ejemplo deseamos saber el número de *Nodos* azules del Grafo para ello en primer lugar debemos definir un modelo basado en el metamodelo del Grafo. Como vemos en el código tenemos dos formas de hacer esto, la primera (linea 01-05) más tradicional y entendible por programadores acostumbrados a programar orientado a objetos corresponde a un simple *for* en el que por cada nodo de todas las aristas del grafo se pregunta si es de color azul, en caso de serlo se suma al contador. La segunda forma (linea 08) es muy similar al lenguaje *OCL* (Object Constraint Language), por todos los nodos del grafo contamos aquellos que sean de color azul.

Como vemos la sintaxis es muy similar a cualquier lenguaje orientado a objetos, podemos manipular y consultar los objetos del modelo. Sin embargo EOL no nos permite la definición de clases, ya que éstas deben de estar definidas en el metamodelo que EOL manipula.

2.3.2. Epsilon Transformation Language

Epsilon Transformation Language (ETL) es un lenguaje de transformación modelo a modelo (*M2M*) basado en reglas. En este proyecto se utiliza ETL para la transformaciones de modelos conceptuales de datos en UML 2.0 en representaciones abstractas de código Cassandra. En el capítulo 1 podemos ver un ejemplo de código ETL.

2.3.3. Epsilon Generation Language

Epsilon Generation Language (EGL) Louis M. Rose (2008) es un lenguaje utilizado para la transformación de modelos a texto (*M2T*) basado en

plantillas.

EGL puede utilizarse para transformar modelos en cualquier tipo de lenguaje, por ejemplo código ejecutable Java, código HTML o incluso aplicaciones completas que comprenden el código en varios lenguajes (por ejemplo, HTML, Javascript y CSS). En este proyecto se utiliza EGL para la generación de código Cassandra Query Language (CQL) a partir de modelos UML 2.0. Cada plantilla de EGL contiene varias secciones. Cada sección puede ser estática o bien dinámica. Una sección estática contiene texto que aparecerá directamente y tal como está en la salida generada por la plantilla. Una sección dinámica comienza con la secuencia '[%'y termina con la secuencia '%]'. La sección dinámica contiene código lenguaje EOL.

2.3.4. EUnit

EUnit Dimitris Kolovos (2014) es el lenguaje utilizado para la definición de pruebas unitarias de la suite Epsilon. EUnit como herramienta sirve para validar que la transformación entre modelos es correcta, además nos permite validar la transformación modelo-texto, entre otros. Para lograr esto EUnit proporciona *assertions* para comparar modelos, archivos y directorios. Las pruebas pueden ser reutilizados con diferentes conjuntos de modelos y datos de entrada. Las diferencias entre los modelos esperados y los reales se pueden visualizar de manera gráfica.

2.4. Cassandra

Esta sección realiza una breve descripción sobre Cassandra Lith A. (2010), un sistema gestor de datos NoSQL basado en columnas. Describiremos brevemente su modelo de datos y la sintaxis de su lenguaje de gestión de datos. Por último, justificaremos el por qué de la elección de Cassandra dentro de este Proyecto Fin de Carrera.

Cassandra es un sistema gestor de datos NoSQL. En general, existen varias diferencias entre estos sistemas NoSQL y los tradicionales sistemas SQL (*Structured Query Language*) , que son:

1. Los sistemas NoSQL no garantizan las propiedades *ACID* (*Atomicity, Consistency, Isolation y Durability*). Por ejemplo, en ciertos casos, Cassandra no garantiza la integridad de los datos, pudiendo contener datos incongruentes. Por ejemplo, una asignatura podría aparecer en la lista de materias cursadas por un alumno, pero dicho alumno podría no aparecer en la lista de alumnos de esa asignatura, lo que sería no consistente. Esto se debe a que sistemas como Cassandra introducen cierta redundancia para favorecer ciertos tipos de consultas.

2. Los sistemas NoSQL no utilizan SQL como lenguaje de consultas (de ahí su nombre). Algunos bases de datos NoSQL utilizan SQL como

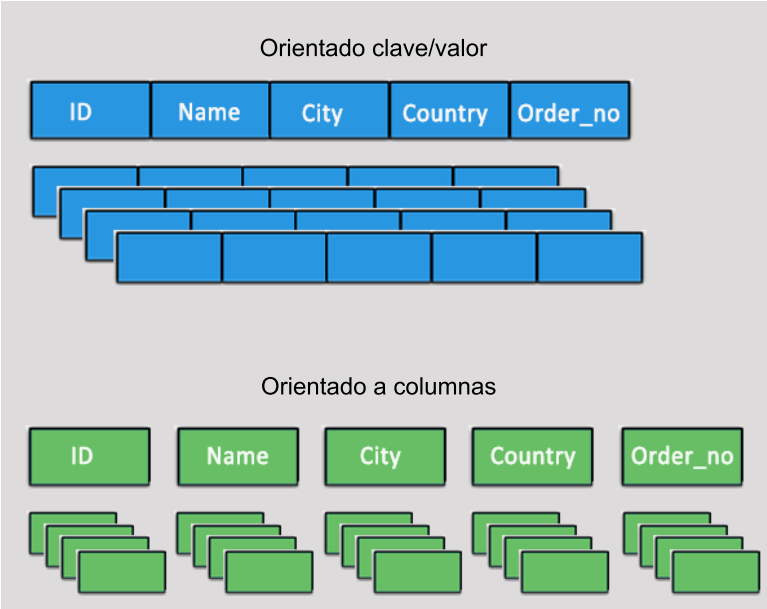

Figura 2.3: Modelo de datos Orientado a Columnas & Clave-Valor

lenguaje de apoyo. Sin embargo, la mayoría utilizan su propio lenguaje de gestión de datos. Por ejemplo, Cassandra utiliza CQL (*Cassandra Query Language*) .

3. Ciertos operadores de los sistemas relacionales, como los *joins*, suelen eliminarse de estos sistemas, ya que al manejar grandes volúmenes de información pueden llegar a sobrecargar el sistema.

4. Escalan horizontalmente y trabajan de manera distribuida. De esta forma, se puede aumentar la capacidad o rendimiento del sistema simplemente añadiendo nuevos nodos.

Dentro del mundo de las bases de datos no relaciones Cassandra es considerada un híbrido de los sistemas *orientados a columnas* y las bases de datos basadas en *clave-valor*. Los basados en columnas tienen como peculiaridad que almacenan los datos en forma de columna. Estos fueron creados para almacenar y procesar grandes cantidades de datos distribuidos en muchas máquinas. Estas columnas son agrupadas en /emphcolumn families. Los sistemas basados en clave-valor son aquellos que asocian los valores a una determinada clave esto permite la recuperación y escritura de información de forma muy rápida y eficiente, esto permite el acceso a los datos de forma rápida, básicamente, utilizan una tabla hash en la que existe una clave única y un puntero a un elemento de datos en particular. En la Figura 2.3 se puede observar de manera gráfica como es la disposición de los datos en cada tipo.

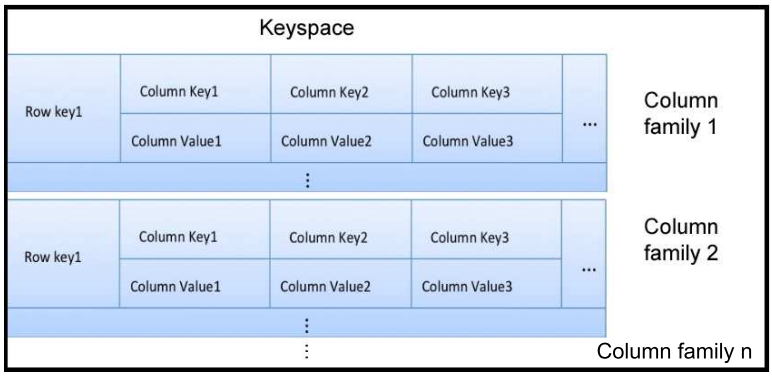

Figura 2.4: Estructura del modelo de datos de Cassandra

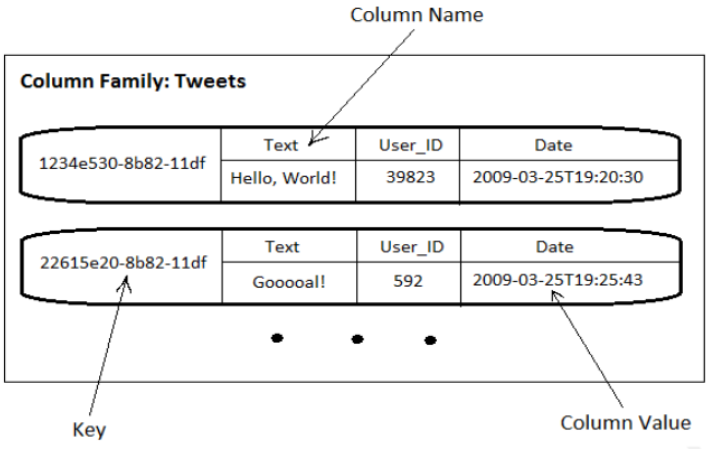

Figura 2.5: Estructura column family

A continuación se explican brevemente el modelo de datos en el cual se basa Cassandra, el modelo de datos es el correspondiente a la Figura 2.4. En Cassandra un keyspace es el contenedor de las column families, es el equivalente a un schema en los sistemas de bases de datos relacionales. La column family es el contenedor de las columns, las column families se guardan en ficheros separados y son ordenadas por su key. Una column es la unidad de almacenamiento básica, está formada por tres campos: Nombre, valor y timestamp. El nombre y el valor se almacena como una matriz de bytes sin procesar y pueden ser de cualquier tamaño. Los tres valores anteriores son introducidos por el cliente, incluido el timestamp. Un ejemplo de la estructura de una column:

Por lo tanto un keyspace puede contener varias column families y una column family a su vez contiene varias columnas. La estructura de una column family queda como se puede observar en el ejemplo de la Figura 2.5.

```
01 DROP KEYSPACE twitter;
02 CREATE KEYSPACE twitter
03 WITH replication = {'class':'SimpleStrategy', 'replication_factor':2};
04
05 USE twitter;
06
07 CREATE TABLE Tweet(
08       UUID text,
09       usernameTw text,
10       body text,
11       PRIMARY KEY(UUID)
12 );
13
14 CREATE TABLE User(
15       username text,
16       password text,
17       followers set<text>,
18       followings set<text>,
19       tweets_written list<text>,
20       PRIMARY KEY(username)
21 );
```

Figura 2.6: Ejemplo código CQL

En cuanto a la sintaxis del lenguaje CQL es muy similar a SQL, CQL contiene sintaxis ya conocida de SQL como INSERT, DELETE o UPDATE. Sin embargo es más limitado que SQL. Como herramienta de manipulación y administración de datos se utiliza *DataStax OpsCenter* que ofrece una interfaz de usuario basada en navegador que sirve para la gestión y monitorización de los cluster Cassandra en una única terminal de administración.

La Figura 2.6 refleja un ejemplo breve de código CQL ejecutable. Las líneas 01-03 sirven para el borrado y la definición de la base de datos ejemplo, en este caso sobre *Twitter*. El resto de líneas definen las column family de la base de datos.

En definitiva las bases de datos basadas en Cassandra son utilizadas cuando es necesaria proporcionar una buena escalabilidad y alta disponibilidad[2]. Además proporciona gran estabilidad en cuanto a la replicación de datos a través de múltiples centros de datos, consiguiendo una menor latencia para sus usuarios y la tranquilidad de que no existan perdidas de datos ante caídas.

[2] `http://goo.gl/7H3EYU` visitada el 10/7/2014

2.5. Sumario

Durante este capítulo se han descrito los conceptos necesarios para lograr comprender el ámbito y el alcance de este proyecto, se ha descrito el caso de estudio planteado en el proyecto. También se ha hablado sobre tecnologías implicadas en el desarrollo del generador de código, así como de Cassandra y su arquitectura, Epsilon los lenguajes utilizados y herramientas utilizadas. El siguiente capítulo describe los primeros pasos para la realización del generador de código, el primer paso consiste en la definición de reglas de transformación entre modelos llamadas Model to Model (M2M). Además se continua el caso de estudio planteado en este capítulo, se define el metamodelo de Cassandra y se definen las reglas de transformación entre modelos UML y Cassandra.

Capítulo 3

Transformación modelo a modelo

Este capítulo describe el primer paso del proceso de transformación de acuerdo al proceso de desarrollo dirigido por modelos. El principal objetivo de esta fase es la transformación de un modelo definido en UML a un modelo en Cassandra. Para ello en primer lugar realizaremos la definición de un metamodelo que defina el modelado en Cassandra. En segundo lugar debemos definir las reglas de transformación para convertir un modelo UML a un modelo en Cassandra, para lograr esto utilizaremos el lenguaje Epsilon Transformation Language (ETL). A continuación se presenta un caso de estudio en el que se van a aplicar las reglas de transformación ETL que se han definido. Finalmente el sumario con las conclusiones de este capítulo.

Índice

3.1. Introducción

A la hora de desarrollar el generador de código necesitamos establecer cuales van a ser los modelos de origen y de salida, en este proyecto de fin de carrera y como se especifica en capítulos anteriores uno de los pasos previos a la construcción del generador de código consiste en transformar un modelo UML en un modelo Cassandra para ello es necesario un meta-modelo que defina Cassandra y un meta-modelo de UML, en cuanto al meta-modelo origen basado en UML 2.0 utilizaremos el que nos proporciona la propia

plataforma Epsilon. Para la construcción del meta-modelo de Cassandra utilizaremos EMF. Este tema es abordado en la siguiente sección de este capítulo.

El siguiente paso consiste en establecer una serie de reglas de correspondencia entre el modelo de entrada y el modelo de salida. Estas reglas han de contemplar los distintos tipos de elementos que pueden aparecer en ambos lenguajes, así como elementos que no aparecen en uno de los dos lenguajes y tiene importancia en el otro. La Sección 3.3 describe dichas reglas y parte del código desarrollado para llevarlas a cabo. En este caso, consiste en establecer reglas de correspondencia entre elementos del modelado UML 2.0 y elementos del modelado Cassandra, para así crear el generador de código Cassandra Query Language (CQL). Para la definición de estas reglas se ha utilizado el lenguaje Epsilon Transformation Language (ETL) y el lenguaje Epsilon Object Language (EOL). Tras implementar estas reglas y comprobar que son funcionales se puede implementar el generador de código pero este tema se trata en el siguiente capitulo.

El presente capítulo describe este proceso de desarrollo. En resumen, la Sección 3.2 describe como se ha realizado el lenguaje de modelado de Cassandra (definición del meta-modelo). La Sección 3.3 presenta las correspondencias definidas ente los elementos UML 2.0 y elementos de Cassandra así como el código necesario para llevar a cabo estas correspondencias, finalmente la sección 3.4 muestra el caso de estudio presentado en el capítulo anterior.

3.2. Metamodelo Cassandra

Como habíamos descrito en el capitulo anterior la base del proceso de transformación entre modelos parte del metamodelo. Previo paso a la definición de las reglas de transformación entre modelos necesitamos definir un metamodelo de origen y un metamodelo de destino. El metamodelo de UML 2.0 utilizado en este proyecto como origen de la transformación es el que nos proporciona Epsilon, dicho metamodelo sigue el estándar de UML 2.0. A parte del metamodelo de UML nos hace falta un metamodelo que defina el lenguaje de modelado de Cassandra. Dicho metamodelo fue proporcionado por Pablo Sánchez Barreiro. Este metamodelo se construyo por medio de la abstracción de las principales características de Cassandra. El metamodelo sufrió algunos cambios respecto al inicial debido a exigencias y variantes que sufrió el proyecto. La Figura 3.1 muestra como se ha definido el lenguaje de modelado de Cassandra con EMF, como vemos se puede diseñar un lenguaje de manera muy similar a la definición de clases en UML. Este tipo de modelo es el origen del proceso de transformaciones entre modelos que queremos crear. A continuación se detallan los aspectos más importantes del metamodelo de Cassandra.

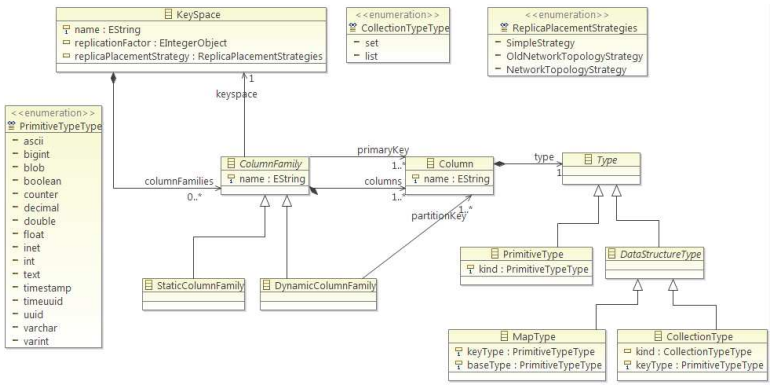

Figura 3.1: Metamodelo Cassandra

De acuerdo con la arquitectura de Cassandra[1], el elemento raíz de cualquier esquema orientado a columnas es el keyspace, el keyspace es el equivalente a una base de datos en el modelo relacional, la meta-clase keyspace cuenta con los meta-atributos: nombre utilizado para denominar el keyspace, Lz (2012) replicationFactor que representa el número de servidores de Cassandra de los que se debe guardar un registro u obtener una respuesta al recuperar algún registro y replicaPlacementStrategy es la estrategia de replicación que se va a tomar, dentro de estas estrategias de replicación encontramos tres tipos DataStax (2014):

1. *SimpleStrategy*: Es la estrategia de replicación utilizada por defecto al crear un keyspace utilizando Cassandra. Es utilizada para clústeres de datacenters simples.

2. *NetworkTopologyStrategy*: Utilizada cuando el clúster es desplegado a través de múltiples data centers. Esta estrategia especifica cuántas réplicas se desean en cada data center.

3. *OldNetworkTopologyStrategy*: Se utiliza para proporcionar retro-compatibilidad con instalaciones de Cassandra antiguas.

A continuación tenemos la meta-clase ColumnFamily equivalente a las tablas en el modelo relacional, dentro de esta meta-clase encontramos dos tipos: las StaticColumnFamily y las DynamicColumnFamily. Recordamos que las static column family son el equivalente a las tablas en el modelo relacional, sin embargo las dynamic column family son utilizadas para la recuperación de datos eficiente, algo similar a las vistas del modelo relacional[2].

[1] `http://www.datastax.com/docs/1.0/cluster_architecture/replication/`

[2] `http://www.datastax.com/docs/1.1/ddl/column_family#dynamic-column-families`

En cuanto a la definición de la primary key, las column families estáticas siguen una definición de claves clásica por lo que en el metamodelo no se refleja, sin embargo las column families dinámicas tienen una definición de la primary key especial. A la hora de definir esta primary key hay que tener en cuenta el orden, en CQL el orden de definición de las claves importa, en la primera columna de la primary key se define la partition key, esta tiene la propiedad de que todas las filas que comparten la misma partition key se almacenan en el mismo nodo físico[3]. Además, la inserción, actualización o eliminación de filas que comparten la partition key para una column family determinada se realizan de forma atómica. Es posible tener una partition key compuesta, es decir, una partition key formada por varias columnas, en CQL esto se define utilizando paréntesis para delimitar el conjunto de partición. En la siguiente columna se define la clustering key utilizada para la recuperación de filas de manera eficiente. En el metamodelo solo tendremos en cuenta las partition key puesto que las columnas que no son definidas como partition key se toman como cluster key. Aunque en siguientes secciones esta información se amplia, el resumen de la definición de column families dinámicas es la siguiente: PRIMARY KEY((partitioning key_1, ... partitioning key_n), clustering key_1 ... clustering key_n).

La siguiente meta-clase es Column, esta meta-clase contiene los valores que se almacenan en las column families, estas columns tienen como meta-atributos el nombre y un tipo de dato. Dentro de los tipos de datos que pueden darse en una columna encontramos dos meta-tipos, PrimitiveType y DataStructureType. PrimitiveType define los tipos primitivos, por ejemplo entero, texto, uuid, etc.. DataStructureType define las colecciones. Estas colecciones pueden ser de dos tipos: MapType o bien CollectionType, ambas colecciones tienen un meta-atributo llamado KeyType para definir el tipo primitivo que utilizan. MapType cuenta con un meta-atributo llamado baseType de tipo primitivo que define el segundo tipo de dato del mapa. Dentro de CollectionType encontramos un meta-atributo llamado kind que define el tipo de colección que vamos a utilizar, esta puede ser o bien tipo set o tipo list. Más adelante se explica que características reúne cada colección y mapa y cuando se utilizan.

3.3. Transformación de Modelo UML a Cassandra

Una vez definido el metamodelo de Cassandra podemos establecer las reglas de transformación entre modelos UML y modelos Cassandra, para ello utilizaremos Epsilon Transformation Language (ETL). Recordamos que el lenguaje ETL es el lenguaje que utiliza Epsilon para la transformación entre modelos basado en reglas. Las reglas de transformación son definidas en Pablo Sánchez (2013). A continuación se detalla cómo se han implementado

[3]http://www.datastax.com/docs/1.1/ddl/indexes

dichas reglas.

3.3.1. Transformación del modelo

El modelo UML se considera el elemento raíz que contiene como es evidente, todos los elementos del modelo, este modelo de datos se transforma en el keyspace para el repositorio de Cassandra. El nombre del keyspace corresponderá al que se haya escogido para el modelo de datos UML. Los keyspaces permiten agrupar entidades tales como column families, columns.. De esta forma, se pueden tener varios Key Spaces en el mismo proyecto independientes entre sí.

3.3.2. Transformación de clases

Se han definido dos reglas ETL que transforman una clase definida en UML a una static column family de Cassandra (el equivalente a una tabla en el sistema de bases de datos relacionales). La primera regla es definida para las clases que tengan un atributo clave definido, para ello se utiliza la propiedad isID de UML, esta propiedad define que ese atributo identifica a la clase de forma única. La segunda regla es para las clases sin atributo clave definido. El proceso de transformación es similar en ambas reglas, en primer lugar se transforma la clase UML a una column family de Cassandra, se asigna la column family al keyspace que le corresponde, se copia el nombre de la clase a la column family y se añade al conjunto de column families del keyspace. La segunda regla es utilizada para las clases sin atributo clave, en este caso se crea un atributo que va a ser la clave de esa column family (ya que esta no tiene), dicha clave tendrá de nombre, el nombre de la clase más el distintivo _ID y será de tipo uuid.

3.3.3. Transformación de atributos

La siguiente regla define como se ha transformado un atributo del modelo UML a una columna del modelo Cassandra. La definición básica es que un atributo UML corresponde a una columna de Cassandra. Para ello definimos una guarda ETL para que la transformación se haga solo de los atributos de las clases y no de los atributos de la relación ya que al importar los modelos UML existen atributos ajenos al modelo de datos que son importados a Epsilon. A continuación se realiza un filtro para evitar la adición de columnas que no pertenezcan al modelo de datos. Una vez hecho esto hay que diferenciar dos tipos de atributos, aquellos cuya multiplicidad sea igual a 1 y aquellos atributos con multiplicidad mayor de 1. En cuanto a los atributos de multiplicidad igual a 1 se realiza la transformación del atributo a columna copiando su tipo de dato. Respecto a los atributos de tamaño mayor de 1 se aplica la siguiente regla: Aquellos atributos que se

hayan definido en el modelo UML como únicos y no ordenados son transformados como tipo set de Cassandra. En caso contrario se definen como tipo list. Una vez transformado el atributo en un set o en una list, en ambos casos se realiza la transformación correspondiente del tipo primitivo UML a Cassandra (viene descrita en la última sub-sección), se realiza una copia del nombre y se añade la columna ya transformada a la column family correspondiente. Finalmente en caso de que el atributo sea clave de la clase se añade a la column family como primary key. El código correspondiente a esta regla es el que se puede observar en la figura 3.2

3.3.4. Transformación de asociaciones

La siguiente regla define como se realiza la transformación de los atributos de las asociaciones entre clases UML. En primer lugar hay que diferenciar como se tratan las asociaciones, existen dos tipos de asociaciones: aquellas asociaciones cuyo extremo tiene una cardinalidad igual a uno y aquellas con una cardinalidad mayor de uno.

Para los extremos con cardinalidad igual a uno se crea una columna que será añadida en la column family del otro extremo de la relación, esta columna nueva es una copia de la columna que es primary key en la column family fuente, el nombre de la nueva columna es la composición del nombre de la column family y del nombre de la primary key, el tipo de la nueva columna es el mismo que el de la columna primary key de la column family fuente. Esto se comprende mejor con el caso de estudio propuesto en la sección siguiente.

Para los extremos con cardinalidad mayor de uno, se crea una dynamic column family. Recordamos que la primary key de este tipo de column family esta compuesta de dos tipos de columnas: La partition key y la cluster key. La transformación utilizada para este tipo de column family es la siguiente: Esta column family dinámica está compuesta de dos columnas, la primera columna juega el papel de partition key y la segunda columna de cluster key, la primera columna se construye de la misma manera que el caso anterior, se copia el nombre y el tipo de la primary key de la column family extremo de la asociación fuente. La segunda columna de la misma manera toma los datos nombre y tipo de la primary key del otro extremo de la asociación. El nombre de la column family será la concatenación del nombre de las column family de ambos extremos de la asociación (primero fuente, segundo extremo fin de la asociación). En resumen la transformación de la primary key en las column family dinámicas es la siguiente:

1. Partition Key: Primera columna (fuente de la asociación).

2. Clustering Key: Segunda columna (extremo de la asociación).

```
------------------------------------------------------
//5.5 Attribute with primitive type transformation
rule Attribute2Column
transform attribute : UML!Property
to column : nosql!Column {
    guard: (""+attribute.qualifiedName).contains("Data::") and attribute.type.isKindOf(UML!PrimitiveType))

    //filtramos para evitar añadir columnas ajenas al modelo de datos
    for(cfamily in kspace.columnFamilies){

        if(attribute.qualifiedName=="Data::"+cfamily.name+"::"+attribute.name){

            //5.5 Attribute with primitive type transformation->Attributes with upper bound = 1
            if(attribute.upper=1){
                //transformacion del atributo en una columna basica
                var type : new nosql!PrimitiveType;
                type.kind=umlType2modelType(attribute.type.name);
                column.type=type;
            }
            //5.5 Attribute with primitive type transformation->Attributes with upper bound > 1
            else if(attribute.upper<>0){
                //transformacion del atributo en un set o list
                var ctype : new nosql!CollectionType;

                if(attribute.isUnique and not attribute.isOrdered)//set
                    ctype.kind=nosql!CollectionTypeType#set;
                else//list
                    ctype.kind=nosql!CollectionTypeType#list;

                ctype.keyType=umlType2modelType(attribute.type.name);
                column.type=ctype;
            }

            column.name=attribute.name;
            cfamily.columns.add(column);

            //5.1 Assignment of keys to classes
            if(attribute.isID)
                cfamily.primaryKey.add(column);
        }
    }
}
------------------------------------------------------
```

Figura 3.2: Regla de transformación atributo-columna

3.3.5. Transformación de tipos primitivos

En cuanto a la transformación de variables de tipo primitivo hemos definido una operación básica de correspondencia entre tipos (no una regla), de esta manera podemos convertir un tipo primitivo UML a su equivalente en Cassandra. Las correspondencias definidas serían las siguientes:

UML	Cassandra
string	text
int	int
date	timestamp
uuid	uuid
float	float
double	double
boolean	boolean
char	varchar

Cuadro 3.1: Equivalencias tipos primitivos UML-Cassandra

3.4. Caso de estudio transformación modelo a modelo

Como se explicaba en el capítulo 2 (sección 2.1), el objetivo consiste en la creación de un generador de código de una versión simplificada de Twitter llamada Twissandra. En esta sección se reproducirán los procesos M2M y M2T en el siguiente capítulo, todo esto bajo el proceso de desarrollo dirigido por modelos. Esta sección está dedicada a describir la transformación del modelo UML de Twissandra a un modelo Cassandra, para ello partimos del modelo UML de la figura 2.1. Una vez establecidas las reglas de transformación entre modelos podemos realizar la generación del código aunque esto se analizará en el siguiente capítulo.

En primer lugar, por cada paquete estereotipado como «dataModel», se crea un nuevo keyspace. El nombre del keyspace será el nombre que se ha definido en el modelo de datos UML. Los atributos restantes de las metaclases del keyspace se establecen en sus valores definidos por defecto en el modelo UML. A continuación, todos los elementos correspondientes de ese paquete se procesan.

A continuación se realiza un marcado del atributo username de la clase User como clave, ya que el parámetro isID del atributo username en el modelo UML está marcado como true. En el caso de las clases FollowingRelationship y la clase Tweet al no tener un atributo marcado como clave generamos dos columnas clave automáticamente para cada clase, llamadas FollowingRelationship_id y tweet_id respectivamente. La clase User del modelo UML es transformada en una column family llamada User. Una vez procesadas las clases UML y transformarlas a su correspondiente column family se proceden a transformar los atributos y asociaciones. De manera similar para aquellos atributos del modelo UML cuya multiplicidad sea igual a uno se realiza una transformación simple, por ejemplo el atributo username y password se transforman en dos columnas Cassandra, ambas del tipo text. Estas

columnas están contenidas en la column family User. De la misma forma se transforman los atributos body y time de la clase Tweet y el atributo timestamp de la clase FollowingRelationship. En el caso del atributo del modelo UML email cuya multiplicidad es mayor de uno y tiene las propiedades isUnique establecida en false y la propiedad isOrdered establecida en false (en el modelo no se puede apreciar pero está configurado así en el modelo UML), se transforma este atributo en una colección de tipo set llamada email cuyo tipo primitivo será text, esta colección estará dentro de la column family User.

En cuanto a las asociaciaciones recordamos que tenemos dos tipos, las de multiplicidad igual a uno y las de multiplicidad mayor de uno. Para la asociación de la clase User cuya multiplicidad es igual uno, se crea una nueva columna llamada user_username (recordemos que username es la clave de la column family user) y esta columna es añadida a la column family Tweet. Para las asociaciones de multiplicidad mayor de uno, por ejemplo la asociación llamada userline se crea una dynamic column family llamada User_userline. A continuación una columna llamada user_username de tipo text es añadida a esta column family. Después una columna llamada tweet_id de tipo uuid es añadida (el atributo tweet_id fue creado en la column family tweet al no tener clave). Las columnas user_username y tweet_id son designadas como primary key, la columna user_username será la partition key y la columna tweet_id será la cluster key.

3.5. Sumario

Durante este capítulo se ha descrito todo el proceso Model to Model (M2M) realizado. En primer lugar se ha descrito la construcción del metamodelo de Cassandra, proceso base para la transformación entre modelos. A continuación se han presentado cuales son las reglas que se han utilizado a la hora de transformar cada uno de los elementos de un modelo UML a un modelo Cassandra. Para lograr este objetivo se han definido una serie de reglas escritas en el lenguaje Epsilon Transformation Language (ETL), de esta manera podemos transformar un modelo UML en un modelo Cassandra. La siguiente sección está dedicada a continuar el caso de estudio desarrollado a lo largo del proyecto relacionado con Twissandra, en esta sección se explican como se transforman los elementos que aparecen en el modelo UML de Twissandra a el modelo de Cassandra.

Capítulo 4

Transformación modelo a texto

El capítulo anterior hizo una descripción del funcionamiento y desarrollo del primer paso del proceso de transformación dirigido por modelos, Model To model (M2M). En este capítulo se presenta el siguiente paso del proceso de transformación dirigido por modelos, dicho paso consiste en la transformación del modelo a código, este paso es mas conocido como Model To Text (M2T). En este capitulo se explica como se ha realizado la implementación del generador de código utilizando para ello el lenguaje Epsilon Generation Language (EGL). Además se describe el funcionamiento y desarrollo de los generadores de código así como de las pruebas realizadas para comprobar su correcto funcionamiento.

Índice

4.1. Introducción

Una vez establecidas las reglas de correspondencia entre ambos modelos empezamos a desarrollar el generador de código. Este capítulo describe el proceso de desarrollo de los generadores de código. El objetivo de esta fase, es generar un repositorio de datos NoSQL funcional a partir de un modelo de datos UML 2.0. El código generado es código Cassandra Query Language (CQL) que puede ser ejecutado en herramientas que soporten dicho lenguaje. Este proceso es descrito en la Sección 3.2 de este capítulo. De manera

análoga al anterior capítulo la Sección 3.3 esta dedicada a aplicar esta transformación modelo-código al caso de estudio sobre Twissandra. A continuación tras implementar el generador de código el siguiente paso consiste en la implementación de una serie de casos de prueba para comprobar el correcto funcionamiento del generador de código. Esto es explicado en la Sección 3.4. Para la implementación de este generador de código se ha utilizado el lenguaje Epsilon Generation Language (EGL).

En resumen, la Sección 3.2 describe como se ha realizado el generador de código. La Sección 3.3 continua el caso de estudio presentado en el capítulo 2, mientras que la Sección 3.4 describe las pruebas que se han realizado para comprobar el correcto funcionamiento del generador de código.

4.2. Generador de código

Esta sección analiza el proceso de creación del generador de código. El desarrollo del generador de código se ha realizado con el lenguaje Epsilon Generation Language (EGL), este lenguaje es utilizado para la generación de código basado en la transformación de modelos, el lenguaje a generar es Cassandra Query Language (CQL). Este lenguaje de consultas es muy similar a SQL, sin embargo existen pequeñas diferencias que se han citado a lo largo del proyecto por ejemplo: la definición de claves, tipos de dato etc.

El proceso de transformación modelo-código es el siguiente. En primer lugar se realiza la definición del keyspace para ello hay que tomar el nombre del keyspace que parte del modelo transformado Cassandra y la estrategia de replicación (ver capitulo 3, sección 2). A continuación se define la creación de una column family en CQL. Tras determinar el bloque de creación de la column family (CREATE TABLE) se definen las columnas de la tabla. Por cada columna de la column family se obtiene el nombre de la columna y se fija el tipo de dato de la columna (primitivo,map,set/list) junto con su nombre. La definición de estas columnas sería la siguiente.

1. *Tipo básico*: "nombre columna-tipo primitivo".

2. *Map*: "nombre columna-map-(tipo primitivo 1,tipo primitivo 2)".

3. *Set/list*: "nombre columna-set-tipo primitivo)".

En la figura 4.1 se muestra parte del código para realizar esto (lenguaje ETL). El resto de código se ha omitido por razones de espacio sin embargo se detalla a continuación.

Tras la definición de la columna se define la primary key. Para la definición de la primary key debemos diferenciar si la column family es dinámica o estática. En caso de ser estática la definición sería la clásica: PRIMARY KEY(ColumnaClave). En el caso de tratarse de una column family dinámica la

```
-----------------------------------------------------
DROP KEYSPACE [%=keyspace.name%];
CREATE KEYSPACE [%=keyspace.name%]
WITH replication = {'class':'[%=keyspace.replicaPlacementStrategy%]', 'replication_factor':[%=keyspace.repl

USE [%=keyspace.name%];

[% for (cf in keyspace.columnFamilies){ tableKey="";%]
CREATE TABLE [%=cf.name%](
[% for (cols in cf.columns){ //este for define cada grupo de columnas
s=cols.name.toString();
for (c in cols.type){

if (c.isTypeOf(PrimitiveType))  //definicion del tipo primitivo
s=s+" "+c.kind.toString();

if (c.isTypeOf(MapType)) //definicion del tipo map
s=s+" Map"+"<"+c.keyType.toString()+","+c.baseType.toString()+">";

  if(c.isTypeOf(CollectionType)) //definicion del tipo set o list
s=s+" "+c.kind.toString()+"<"+c.keyType.toString()+">";

}
);
[%}%]
-----------------------------------------------------
```

Figura 4.1: Regla de transformación Primary Key

construcción modelo-código es distinta, la más frecuente es la siguiente: PRIMARY KEY(partitioning key, clustering key_1 ... clustering key_n) Sin embargo hay que tener en cuenta que en la construcción y según la documentación de CQL también es posible tener una partition key compuesta, es decir, una partition key formada por varias columnas, para realizar esto se utilizan paréntesis y así delimitamos el conjunto de partición. Quedando una posible definición de la primary key de la siguiente manera: PRIMARY KEY((partitioning key_1, ... partitioning key_n), clustering key_1 ... clustering key_n). Este proceso se repite por cada column family del modelo.

Una vez definido el código EGL podemos poner en funcionamiento el generador de código. El proceso completo sería el siguiente: En primer lugar creamos un modelo UML con cualquier herramienta de modelado, por ejemplo el modelo UML de Twissandra ha sido realizado con la herramienta Magic Draw. A continuación transformamos este modelo UML al modelo Cassandra con los métodos de transformación definidos en el capítulo anterior, una vez obtenido este modelo de Cassandra podemos poner en funcionamiento el generador de código, este proceso con el caso de estudio de Twissandra es descrito en la siguiente sección. Con el código ya generado podemos crear el repositorio de datos Cassandra.

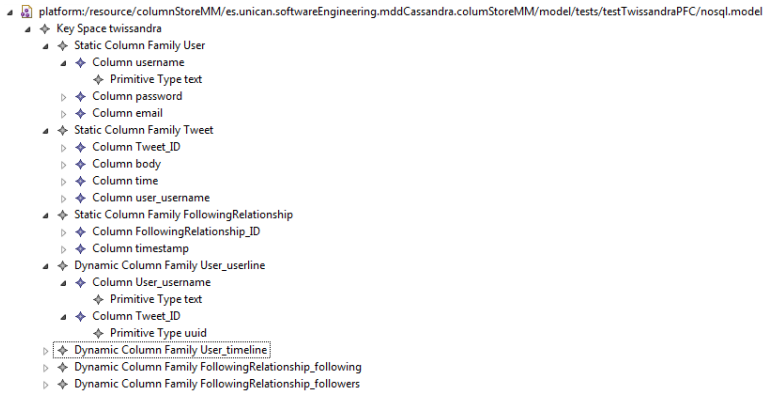

Figura 4.2: Modelo UML Twissandra

4.3. Caso de estudio Twissandra

Una vez definido el funcionamiento del generador de código podemos continuar con el caso de estudio planteado en el segundo capítulo sobre Twissandra. Como se presentó anteriormente el objetivo de este caso de estudio es la generación de código CQL a partir de un modelo UML 2.0. Para ello definimos una serie de reglas de transformación entre modelos y obtuvimos un modelo de Cassandra a partir de un modelo UML que representaba una versión simplificada de Twitter llamada Twissandra. En esta sección se presenta el resultado de la transformación del modelo de Twissandra a código.

Tras obtener el modelo Cassandra de Twissandra (figura 4.2) y tras la definición del generador de código se pone en funcionamiento la transformación modelo-código. El resultado de la transformación se detalla a continuación.El código resultante tras ejecutar el generador de código es el siguiente (figura 4.3).

Como vemos en el código las tres primeras líneas son para borrar el keyspace en caso de que existiera, crearlo y configurar la arquitectura de replicación según estaba especificado en el modelo Cassandra. A continuación se conecta la sesión con el keyspace correspondiente. Por cada column family (sea estática o dinámica) se genera un bloque CREATE TABLE. Cada columna de la column family es transformada en una columna de la tabla. La transformación como se contaba en la sección anterior es trivial, sin embargo el problema del generador de código surge a la hora de definir la estructura de la primary key. Las column family estáticas siguen una definición de claves tradicional, sin embargo las column families dinámicas tienen que tener en cuenta que las claves de partición (partition keys) pueden ser compuestas (aunque en este ejemplo no existen). Por ejemplo el caso de user_userline las columnas user_name y tweet_id son designadas como primary key, user_username será la clave de partición.

4.4. Pruebas

Una vez implementado el generador de código la siguiente tarea consiste en comprobar que el código generado funciona correctamente. Para ello creamos, una serie de pruebas unitarias que permitan comprobar que el funcionamiento de los generadores de código es correcto para un conjunto de modelos de entrada.

Estas pruebas unitarias se han implementado en EUnit, el lenguaje de definición de pruebas de la suite Epsilon. EUnit funciona de una manera muy similar a JUnit, pero aplicado a los lenguajes de la suite Epsilon, como EGL. Utilizamos EUnit para comprobar que el funcionamiento del generador de código es correcto, para ello se diseñan una serie de casos de prueba y se crea la salida esperada de cada uno de esos casos de prueba de forma manual. A continuación, se ejecuta el caso de prueba creado en EUnit y se comprueba que la salida generada coincide con la esperada, que es la creada manualmente. Además comprobamos con casos específicos que la cobertura del código es del 100

El único problema surge por un problema de defecto en la herramienta EUnit, la función de comparación de ficheros llamada assertEqualFiles obliga a que ambos ficheros sean estrictamente iguales por lo que para crear casos de prueba tanto el código de entrada de prueba como el código generado a testear han de ser iguales, espacios y saltos de línea incluidos.

4.5. Sumario

Durante este capítulo se ha descrito el proceso de desarrollo del generador de código. Tras la breve introducción de este capitulo se ha presentado parte del código del generador de código así como la sintaxis de EGL, lenguaje utilizado para construir este generador. Además se ha detallado como se ha realizado este generador de código. A continuación se ha detallado el proceso de transformación del caso de estudio introducido en el capítulo dos de Twissandra. Finalmente se han presentado las pruebas realizadas así como la herramienta utilizada EUnit.

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

USE twissandra;

CREATE TABLE User(
       username text,
       password text,
       email set<text>,
       PRIMARY KEY(username)
);

CREATE TABLE Tweet(
       Tweet_ID uuid,
       body text,
       time timestamp,
       user_username text,
       PRIMARY KEY(Tweet_ID)
);

CREATE TABLE FollowingRelationship(
       FollowingRelationship_ID uuid,
       timestamp timestamp,
       PRIMARY KEY(FollowingRelationship_ID)
);

CREATE TABLE User_userline(
       User_username text,
       Tweet_ID uuid,
       PRIMARY KEY((User_username),Tweet_ID)
);

CREATE TABLE User_timeline(
       User_username text,
       Tweet_ID uuid,
       PRIMARY KEY((User_username),Tweet_ID)
);

CREATE TABLE FollowingRelationship_following(
       FollowingRelationship_FollowingRelationship_ID uuid,
       username text,
       PRIMARY KEY((FollowingRelationship_FollowingRelationship_ID),username)
);

CREATE TABLE FollowingRelationship_followers(
       FollowingRelationship_FollowingRelationship_ID uuid,
       username text,
       PRIMARY KEY((FollowingRelationship_FollowingRelationship_ID),username)
);
```

Figura 4.3: Código resultante Twissandra

Capítulo 5

Sumario y Trabajos Futuros

Este capítulo resume el trabajo realizado durante la elaboración del presente Proyecto Fin de Carrera, además se describen brevemente las lecciones, experiencias personales aprendidas y posibles trabajos futuros.

Índice

5.1. Sumario

Esta memoria de Proyecto Fin de Carrera ha descrito el proceso de desarrollo de un generador de código Cassandra bajo el desarrollo dirigido por modelos.

Como hemos visto el Desarrollo Dirigido por Modelos aporta múltiples ventajas a los desarrolladores software, la automatización de los procesos de producción nos permite crear software más rápido llegando a considerar a este paradigma como la verdadera industrialización de la producción de software, esto permite a las empresas ahorro en tiempo y costes. La base de la utilización de Cassandra se sostiene en aspectos como la disponibilidad o el manejo de gran cantidad de datos, aspectos que no son facilitados mediante la utilización de bases de datos relacionales tradicionales. Además dentro de las bases de datos no relacionales orientadas a columnas Cassandra es el sistema utilizado por excelencia. La utilización de modelos UML respecto a modelos especificados en Cassandra a la hora de crear una base de datos no relacional proporciona una abstracción para aquellos desarrolladores que no estén muy familiarizados con el modelado de bases de datos no relacionales. La utilización de modelos diseñados en UML proporciona las siguientes ventajas: (1) UML es un lenguaje de modelado bien conocido

por toda la comunidad. (2) La automatización de estos procesos nos permite crear software más rápido, más fiable y de mayor calidad lo que nos lleva a mantener buenas prácticas.

A nivel de implementación, el desarrollo del generador de código fue laborioso en términos de adquisición de conceptos ya que el plan de estudios de la carrera de Ingeniería Informática aporta pocos conocimientos en este área. Para ello en primer lugar se estudiaron conceptos relacionados con el modelado de lenguajes Kleppe (2009) para entender y poder trabajar con el metamodelo de Cassandra, a continuación todo lo vinculado con el Desarrollo y la Ingeniería Dirigida por Modelos, técnicas de transformación (M2M,M2T), ventajas etc.. Finalmente el estudio de todos los lenguajes y herramientas que ofrece la plataforma de Epsilon (ETL, EGL, EOL, EUnit..). Una vez dominados estos conceptos se pusieron en práctica todas las nociones aprendidas con una serie de ejemplos sencillos propuestos por el director del proyecto que han sido descritos a lo largo de esta memoria.

Tras estas tareas de estudio, se procedió a la realización del principal objetivo del proyecto: la creación de un generador de código Cassandra que transforma modelos UML a modelos Cassandra. Para ello en primer lugar se hicieron una serie de cambios en el metamodelo que nos proporcionaron de Cassandra mediante EMF. A continuación se definieron las reglas de correspondencia entre modelos UML y Cassandra y se desarrollaron mediante el lenguaje ETL. Seguidamente y tras realizar el proceso de transformación entre modelos (M2M), se desarrollaron las plantillas para la transformación modelo-código (M2T), para lograr esto se crearon plantillas para generar código CQL, de esta manera obtuvimos el código necesario para crear repositorios de datos en Cassandra. Por último se realizaron una serie de casos de prueba con la herramienta EUnit para comprobar el correcto funcionamiento del generador de código.

La siguiente sección describe las experiencias personales vividas a lo largo del proyecto.

5.2. Experiencia personal

Esta sección describe las experiencias personales acontecidas a lo largo del proyecto. En primer lugar las tareas de aprendizaje de lenguajes como EOL, ETL o EGL no resultaron muy costosas sin embargo siempre cuesta adentrarse en materias desconocidas ya que a pesar de que EOL por ejemplo es similar a OCL (Object Constraint Language) son conceptos que no se ven mucho durante la carrera.

En cuanto a la tarea de desarrollo del generador de código se utilizó la herramienta Epsilon que ofrece funcionalidades que facilitan el trabajo a la hora de crear generadores de código, sin embargo el mayor problema de estas herramientas es que al ser de reciente creación el numero de errores que

surgen durante el desarrollo son molestos y numerosos, por ejemplo existen errores sin identificar durante la ejecución de los casos de prueba. También surgen dificultades a la hora de poner en marcha el generador de código ya que algunos parámetros de configuración de Epsilon no son descritos en los manuales, a pesar de que Epsilon proporciona a los usuarios manuales y tutoriales completos y didácticos pero a nivel elemental. A pesar de algunos infortunios o problemas a lo largo del desarrollo el resultado del trabajo es satisfactorio.

El paradigma del Desarrollo Dirigido por Modelos en mi opinión será una materia básica a la hora de estudiar tecnologías y herramientas que pueden facilitar la vida a las empresas. Personalmente creo que estas tecnologías utilizadas para la automatización de construcción software son parte del futuro y serán materias troncales para futuros alumnos que estudien Ingeniería Informática, ya que permiten a las empresas reducción de costes, ahorro de tiempo y reutilización de componentes tres pilares clave en materia de optimización de los recursos de una empresa software.

5.3. Trabajos futuros

Este proyecto de fin de carrera ha cubierto prácticamente en su totalidad los objetivos planteados desde el principio, es posible que el proyecto deje algún aspecto por completar para que el trabajo presentado tenga la apariencia de un producto profesional sin embargo hay propiedades en las que la herramienta Epsilon ha puesto muchas dificultades, por ejemplo: muchos bugs de carácter desconocido, dificultades para el empaquetado del generador de código, funcionalidades sin cubrir por la herramienta EUnit.. Los problemas sin cubrir son comprensibles ya que al ser una herramienta desarrollada por terceros estos defectos están fuera de nuestro alcance por lo que difícilmente pueden ser subsanados. Además hay que tener en cuenta que Epsilon es una herramienta aún joven. Otro posible futura funcionalidad encontrada es que Epsilon ofrezca facilidades para el empaquetado del generador de código de manera que se pueda crear un plug-in que se integre en Eclipse, de esta manera cualquier usuario de Eclipse con la instalación sencilla del plug-in podría ejecutar este generador de código de manera muy sencilla. Otra de las tareas que se podría desarrollar es la integración del generador de código con Apache Cassandra esto podría resultar útil a los desarrolladores ya que con el diseño UML del sistema a implementar se podría generar el repositorio de datos de manera muy sencilla.

Appendix A

Descripción de los contenidos del CD adjunto

El contenido adjunto al CD de esta memoria contiene los siguientes archivos descritos a continuación:

1.

2.

3.

Bibliografía

Budinsky, F. e. a. (2009). *EMF: Eclipse Modeling Framework*. Addison-Wesley.

Cáceres, D. A. (2008). Desarrollo de software para robots de servicio: Un enfoque dirigido por modelos y orientado a componentes.

Clay, K. (2013). Amazon.com goes down, loses 66,240 dollars per minute.

DataStax (2014). Apache cassandra 2.0 documentation.

Dave Steinberg, Frank Budinsky, M. P. E. M. (2008). *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional.

Dimitris Kolovos, Louis Rose, A. G.-D. R. P. (2014). *The Epsilon Book*. The Eclipse Foundation.

Elmasri, R. and Navathe, S. (2010). *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition.

George, L. (2011). *HBase: The Definitive Guide*. O'Reilly.

Kleppe, A. (2009). *Software Language Engineering: Creating domain-specific using metamodels*. Addison-Wesley.

Lakshman, A., M. P. (2010). Cassandra: A decentralized structured storage system. In *ACM SIGOPS Operating System Reviews*, volume 44, pages 35–40.

Lith A., M. J. (2010). Investigating storage solutions for large data. In *A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data*, pages 23–24.

Louis M. Rose, Richard F. Paige, D. S. K. F. A. C. P. (2008). The epsilon generation language. In *4th European Conference, ECMDA-FA*.

Lz, D. D. (2012). Bases de datos no relacionales (nosql deustotech).

Marco Brambilla, Jordi Cabot, M. W. (2012). *Model-Driven Software Engineering in Practice (Synthesis Lectures on Software)*. Morgan & Claypool Publishers.

Olivé, A. (2007). *Conceptual Modeling of Information Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

OMG (2007). MOF 2.0/XMI Mapping V. 2.1.1. Technical report, Object Management Group.

Pablo Sánchez, M. Z. (2013). On the transformation of uml attributes and associations to column-oriented nosql systems.

Plugge, E., H. T. M. P. (2010). *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress.

Stonebraker, M. (2010). sql databases v. nosql databases. In *S. Communications of ACM*, volume 53, pages 10–11.