Inferencia del esquema en bases de datos NoSQL a través de un enfoque MDE

Severino Feliciano Morales, Jesús García Molina, and Diego Sevilla Ruiz

Facultad de Informática, Universidad de Murcia
Campus Espinardo, Murcia, España
{severino.feliciano,jmolina,dsevilla}@um.es

Resumen La ausencia de un esquema de datos explícito (*schemaless*) es una de las características que más ha contribuido al éxito de las bases de datos NoSQL, dado que proporciona flexibilidad y favorece la evolución. Sin embargo, el conocimiento del esquema de datos subyacente es necesario para los desarrolladores NoSQL, y también para herramientas que manejan este tipo de bases de datos, por ejemplo para realizar análisis sobre los mismos. En este artículo se presenta una solución de ingeniería inversa para inferir el esquema de bases de datos NoSQL, que tiene en cuenta el versionado de los datos. El esquema extraído se representa en forma de un modelo de entidades basado en la sintaxis y semántica de los modelos de clases de UML. La solución se ha implementado con tecnología MDE (*Model-Driven Engineering*) para conseguir una independencia del tipo de base de datos y los beneficios de abstracción y automatización propios de MDE.

Key words: Ingeniería Inversa de Datos, Bases de Datos NoSQL, Inferencia del Esquema, Ingeniería Dirigida por Modelos, MDE, JSON

1. Introducción

Las bases de datos NoSQL (*Not SQL*) se han consolidado como el sistema de almacenamiento más apropiado para atender las demandas de las nuevas aplicaciones que deben manejar grandes volúmenes de datos complejos y un mayor tráfico en la red. El acrónimo NoSQL se utiliza para agrupar los diferentes paradigmas de bases de datos que han surgido para paliar las limitaciones de los sistemas relacionales ante los nuevos requisitos. En realidad, aunque el término NoSQL no se refiere a un conjunto bien definido de propiedades, sí es cierto que la mayoría de bases de datos consideradas NoSQL tienen algunas propiedades comunes: no utilizan el lenguaje SQL, no existe un esquema de datos explícito, suelen ser de código abierto (*open-source*) y su diseño está determinado por la necesidad de ejecutarse sobre clusters de ordenadores [4,5].

La ausencia de un esquema de datos explícito (*schemaless*) es una de las características NoSQL más atractivas, ya que proporciona dos importantes beneficios: facilita los cambios en la estructura de los datos y permite manejar datos no uniformes. Sin embargo, el hecho de que el esquema de datos esté implícito en

el código de las aplicaciones conlleva algunos problemas entre los que destacan los dos siguientes [5]. Por un lado, es necesario analizar el código para deducir el esquema que podrá incluir diferentes versiones de las entidades. Por otro lado, las herramientas que ofrecen utilidades para bases de datos NoSQL (por ejemplo, validaciones que aseguren un uso consistente de los datos por todas las aplicaciones) requieren conocer el esquema que rige la estructura de los datos.

En este trabajo se presenta un proceso de ingeniería inversa para la inferencia del esquema de bases de datos NoSQL a partir de los datos almacenados. El enfoque es aplicable a modelos de datos orientados a la agregación en los que se basan las soluciones NoSQL más extendidas [5]: *orientadas a documento*, *clave-valor* y *orientadas a columnas*. Nuestro enfoque se ha implementado con técnicas de la Ingeniería de Software Dirigida por Modelos (*Model Driven Engineering*, MDE) con el fin de aprovechar las ventajas de los modelos y transformaciones de modelos para crear arquitecturas reutilizables, extensibles e interoperables [7]. Un esquema es inferido a partir de una transformación modelo-a-modelo que parte de un modelo que representa datos NoSQL en formato JSON y genera un modelo que representa esquemas de datos orientados a la agregación. Los modelos del esquema obtenidos son entrada a transformaciones modelo a texto que proporcionan a los desarrolladores una representación gráfica y textual del esquema. Los esquemas son visualizados gráficamente aprovechando las herramientas disponibles para visualizar metamodelos Ecore [8] como diagramas de clase (por ejemplo Sirius[1]). La estrategia transformacional aplicada ilustra el papel de los modelos como *lingua franca* que permite la integración de tecnologías y herramientas existentes y como representación abstracta del conocimiento extraído en un proceso de ingeniería inversa.

La inferencia de esquemas de documentos JSON ha sido tratada previamente en el dominio de servicios web [1]. Nuestro trabajo supone una contribución significativa dado que, según nuestro conocimiento, es el primer enfoque publicado sobre inferencia de esquemas de bases de datos NoSQL que descubre las distintas versiones de los datos almacenados y las relaciones de agregación y referencia entre ellos, y genera tanto una representación textual como gráfica del esquema. Además, es un ejemplo ilustrativo de la utilidad de MDE en ingeniería inversa y en integración de herramientas.

La herramienta desarrollada ha sido validada para las bases de datos de documentos MongoDB[2] y CouchDB[3], pero el enfoque podría ser aplicado a otros sistemas NoSQL basados en datos agregados.

El artículo está organizado de la siguiente manera. La siguiente sección introduce la noción de sistemas NoSQL orientados a la agregación e ilustra cómo el formato JSON representa datos NoSQL. La sección 3 presenta un esquema de la solución MDE propuesta y describe los dos metamodelos definidos. La sección 4 explica el proceso de ingeniería inversa ideado para inferir el esquema a partir de los datos JSON. La sección 5 señala cómo se ha llevado a cabo la implementación

[1] `http://eclipse.org/sirius/`.
[2] `http://www.mongodb.org/`.
[3] `http://couchdb.apache.org/`.

y cómo se ha evaluado su eficiencia. En la sección 6 se contrasta nuestra propuesta con otras soluciones de extracción de esquemas a partir de datos JSON y finalmente la sección 7 comenta las conclusiones y futuros trabajos.

2. Sistemas NoSQL basados en la agregación

El método *Domain-Driven Design* [2] introdujo el término "agregado" para referirse a una estructura de objetos determinada por una entidad[4] raíz que agrega recursivamente a otras entidades, dando lugar a una jerarquía de agregación. Un agregado (instancia de una jerarquía de agregación) es manipulado como una única entidad, ya que el objeto raíz controla el acceso a los objetos que agrega con el fin de asegurar la consistencia de los cambios. En realidad esta noción de agregado proviene de dotar de una semántica más precisa a la conocida relación de "agregación" del modelado de datos, que encontramos, por ejemplo, en los diagramas de clases de UML. *Agregación* y *referencias* entre entidades son los dos tipos de asociaciones comúnmente utilizadas en el modelado de datos. La Figura 1 muestra un sencillo diagrama de clases UML que corresponde a un esquema de datos con agregados y referencias. Los libros (entidad *Book*) agregan a sus autores (entidad *Author*) y tienen una referencia a su editorial (*Publisher*). Además, cada entidad tiene un conjunto de atributos o propiedades.

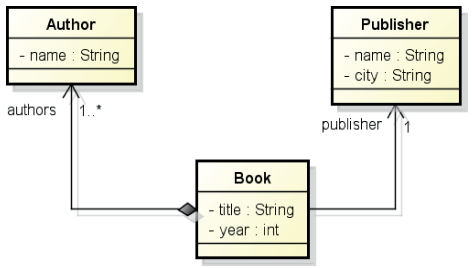


Figura 1. Ejemplo de esquema con asociaciones de agregación y referencia.

Frente a las referencias (*joins*) entre tablas a través de claves ajenas de los sistemas relacionales, la mayoría de sistemas NoSQL organizan los datos como agregados, y las referencias se limitan en la medida de lo posible, dado que implican dependencias entre datos y limitan el tratamiento paralelo y distribuido de los datos. En [5] se evidencia que los tres tipos de sistemas NoSQL más extendidos (orientados a documentos, clave-valor y orientados a columnas) pueden ser considerados conformes a un modelo de datos orientado a la agregación. Los sistemas clave-valor proporcionan el modo más simple de almacenamiento: una colección de pares clave-valor, similar a la estructura *map* que encontramos en

[4] Utilizaremos el término *entidad* para referirnos a objetos persistentes.

muchos lenguajes de programación. En el caso de los sistemas orientados a documentos, en los pares clave-valor, el valor es un documento que representa a un tipo de entidad raíz (*Book* y *Publisher* en el ejemplo anterior). En los sistemas orientados a columnas el almacenamiento está formado por una colección de filas (*rows*) cada una de las cuales está formada por una clave (*row key*) y una familia de columnas (*column family*) formada por un conjunto de pares clave-valor. En cualquier caso, los datos almacenados en la mayoría de estos sistemas pueden ser analizados de un modo uniforme dado que son colecciones de pares clave-valor y los valores pueden ser, a su vez, colecciones de pares clave-valor (agregación).

A diferencia de los sistemas relacionales, los sistemas NoSQL no requieren de la definición de un esquema de los datos que restrinja la estructura de los mismos. Esta ausencia de esquema (*schemaless*) proporciona flexibilidad y favorece la evolución. Por una parte, es posible disponer de distintas versiones de una determinada entidad con diferentes propiedades (*no uniformidad de datos*). Por ejemplo, el traductor de un libro o la organización a la que pertenece un autor, no se debería registrar para todos los libros y autores, respectivamente. Además la estructura de los datos puede evolucionar a lo largo del tiempo, por ejemplo, los datos se pueden enriquecer con propiedades requeridas por nuevas versiones de las aplicaciones.

Como hemos indicado, esta naturaleza *schemaless* es una característica que ha influido en gran medida al éxito de los sistemas NoSQL. Sin embargo, tanto los desarrolladores de aplicaciones para estos sistemas como las herramientas que proporcionan utilidades sobre ellos necesitan conocer el esquema implícito. Los programadores deben escribir sus aplicaciones de bases de datos teniendo en mente cómo están organizados los datos y entre los ejemplos de herramientas que han de inferir el esquema encontramos a Drill[5], que es capaz de generar consultas SQL para cualquier sistema NoSQL.

2.1. JSON para datos NoSQL

JSON (*JavaScript Object Notation*) es un formato textual estándar para representar datos estructurados cuya aceptación crece continuamente debido que es más simple y legible que XML. Los documentos JSON no conforman a un esquema sino que incluyen tanto los datos como su estructura, como se puede observar en la Figura 2. Por ello, la mayoría de sistemas NoSQL utilizan este formato para representan la información que almacenan y ofrecen soporte para exportar/importar datos JSON.

Un objeto JSON está formado por un conjunto de pares clave-valor. El tipo de un valor JSON puede ser de un tipo primitivo (*Number*, *String*, *Boolean*), un objeto, un array de valores o *null*, para indicar que no existe valor para una clave. Dado que un objeto puede tener pares cuyo valor sea otro objeto, es posible representar agregados. Por otro lado, una referencia se puede expresar como un par cuyo valor es igual al valor de otra propiedad en otro objeto. Por tanto, una base de datos NoSQL basada en agregación puede ser creada como una colección

[5] `http://drill.apache.org/`.

de objetos JSON (entidades). De hecho, los sistemas que gestionan estas bases de datos soportan la importación y exportación de datos en un formato JSON propio de cada sistema. Por ejemplo, MongoDB exporta cada colección de la base de datos como un *array* de objetos JSON y CouchDB exporta un único objeto que contiene todos los objetos de la base de datos.

En nuestro caso, se han normalizado los datos JSON exportados en un formato común: un *array* con un objeto por cada tipo de entidad que a su vez incluye un *array* con todos los objetos de ese tipo. Por ejemplo, la Figura 2 muestra una base de datos simple para una versión extendida del esquema de la Figura 1 en la que se han incluido varias versiones de las entidades. El *array* tiene tres objetos, uno por cada tipo de entidad raíz en el esquema: *publisher*, *book* y *journal*. Nótese que los autores y contenidos están embebidos en los objetos *book*. El primer objeto de tipo *book* agrega autores (campo *authors*) y contenidos (campo *content*), y a su vez *authors* agrega a la empresa en la que trabaja el autor (campo *company*). También se puede observar que los datos no son uniformes, existiendo varias versiones para cada entidad. Por ejemplo en el caso del documento *book* los autores también se registran como una lista de nombres.

El ejemplo muestra cómo se pueden expresar referencias entre entidades: dos objetos *publisher* incluyen una referencia a las revistas editadas por esa editorial (campo *journals*) y dos objetos *book* incluyen una referencia a la editorial (campo *publisher_id*). Una referencia con cardinalidad 0..1 es registrada como un entero, un *string* o el valor `null`, por ejemplo *"publisher_id":"AddisonWesley"* en el primer objeto *book*, mientras que una referencia 0..* es registrada como un array de enteros o *strings* o bien el valor `null`, por ejemplo *"journals":[1,2]* en el primer objeto *publisher*. Nótese que las referencias no se explicitan con ningún tipo de construcción sintáctica de JSON, sino que deberían ser inferidas mediante un análisis de los datos. En bases de datos NoSQL hay convenciones como usar el sufijo *_ref* como nombre de una clave cuyo valor es una referencia o bien el objeto {*_ref:nombre-entidad, _id:id-referencia*} pero dependen de la base de datos, y los programadores no están obligados a aplicarlas. Obsérvese también que hay objetos *publisher* sin el campo *journals* y libros sin el campo *publisher*.

3. Metamodelos JSON y Esquema NoSQL

La solución MDE ideada consta de tres etapas como se observa en la Figura 3. Primero se realiza una inyección de los datos JSON en un modelo conforme al metamodelo JSON (Figura 4). Para ello, se ha utilizado un inyector de modelos JSON que hemos generado con una herramienta de definición de lenguajes específicos del dominio (DSL) textuales (*language workbench*). En segundo lugar, se ejecuta una transformación modelo a modelo (m2m) que implementa el proceso de inferencia del esquema, la cual tiene como entrada el modelo JSON inyectado y genera un modelo conforme al metamodelo *NoSQL Schema* (Figura 5). A partir de este modelo se generan una representación textual y gráfica del esquema de la base de datos por medio de dos sencillas transformaciones modelo a texto (m2t). En el caso del diagrama, la transformación establece una

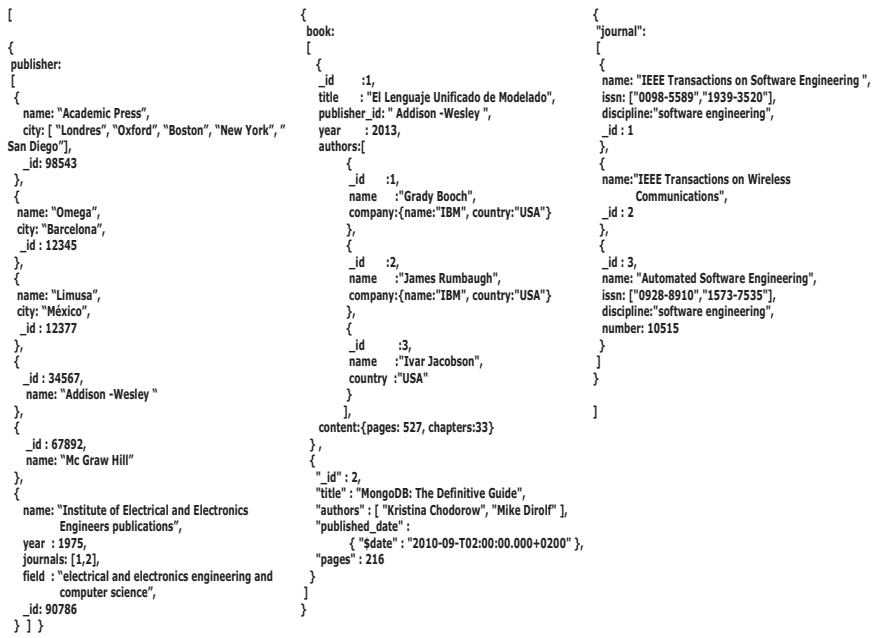


Figura 2. Ejemplo de base de datos NoSQL.

correspondencia entre el metamodelo *NoSQL Schema* y el metamodelo Ecore con el fin de aprovechar las herramientas que muestran modelos Ecore como diagramas de clases UML. Antes de explicar en detalle el proceso de inferencia del esquema en la siguiente sección, describimos los dos metamodelos que hemos definido. La implementación de la solución será descrita en la Sección 5.

La Figura 4 muestra el metamodelo JSON. Se trata de un metamodelo muy sencillo que refleja el bien conocido *mapping* entre una gramática y un metamodelo, en este caso la gramática JSON. Hemos supuesto que un documento (metaclase *Document*) puede contener cero o más elementos (*Element*) que pueden ser objetos (*Object*) o arrays (*Array*). Un objeto (*Object*) puede contener uno o varios pares "clave-valor" (*Pair*) que tienen un atributo *nombre* y contienen un valor (*ValueObject*). Este valor puede ser una cadena de caracteres (*StringJSON*), un booleano (*BooleanJSON*), un entero (*NumberJSON*) o un real (*FloatJSON*) o bien otro elemento.

La Figura 5 muestra el metamodelo NoSQL Schema ideado para representar esquemas de sistemas NoSQL orientados a la agregación. Un esquema (metaclase *NoSQLSchema*) está formado por una colección de entidades (*Entity*) de las que existen varias versiones (*EntityVersion*). Cada versión es definida por una colección de propiedades (*Property*) que tienen un nombre y pueden ser de dos tipos, según representen atributos (*Attribute*) o relaciones entre entidades

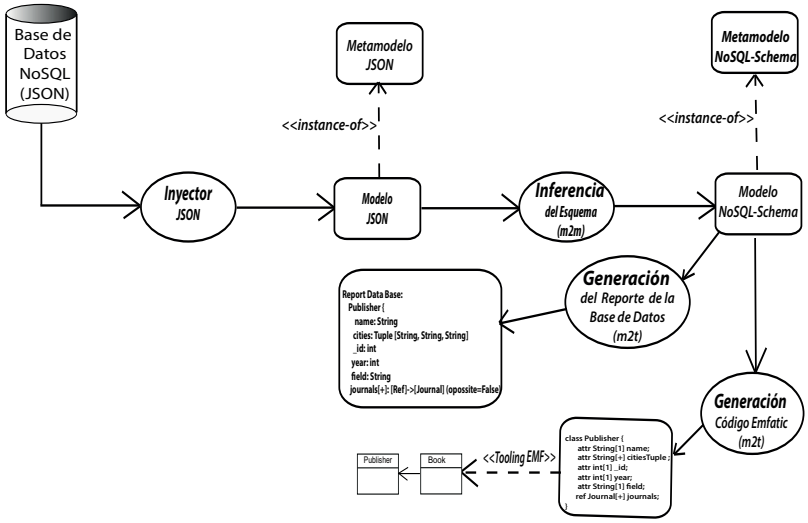


Figura 3. Esquema de la solución MDE propuesta.

(*Association*). Un atributo tiene un tipo (*Type*) que puede ser primitivo (*PrimitiveType*) o una tupla (*Tuple*) que denota una colección de tipos primitivos. Por otro lado, una asociación tiene una cardinalidad (atributos *lowerBound* y *upperBound*) y puede ser de dos tipos: i) un documento embebido (*Aggregation*), es decir, agregado a la entidad que contiene la propiedad, o ii) una referencia (*Reference*) a otra entidad. Nótese que una referencia se conecta a una entidad (*[1..1]refTo*), mientras que en el caso de una agregación se conecta a una o más versiones de entidades (*[1..*]refTo*) dado que un objeto embebido puede agregar una lista de objetos con diferente estructura. Se incluye la referencia reflexiva *rol opposite* para establecer que una *Reference* es la inversa de otra.

4. Proceso de Ingeniería Inversa

La obtención de un modelo NoSQL Schema a partir de un modelo JSON implica la inferencia de los elementos que componen un esquema de datos en los sistemas NoSQL considerados: *Entidades*, *Atributos*, *Referencias* y *Agregaciones*. El proceso de inferencia recorre todos los objetos JSON del modelo de entrada que no están embebidos en otro objeto (entidad raíz) y analiza sus propiedades para ir identificando el esquema como se explica abajo después de definir el concepto de tipo de un objeto JSON.

El tipo de un objeto JSON está formado por una lista de pares *nombre* y *tipo-valor* obtenida de los pares JSON del objeto al asociar cada nombre de clave con el tipo de su valor. Cuando el valor es un objeto o un array de objetos se recorre de manera recursiva los pares de los objetos anidados para

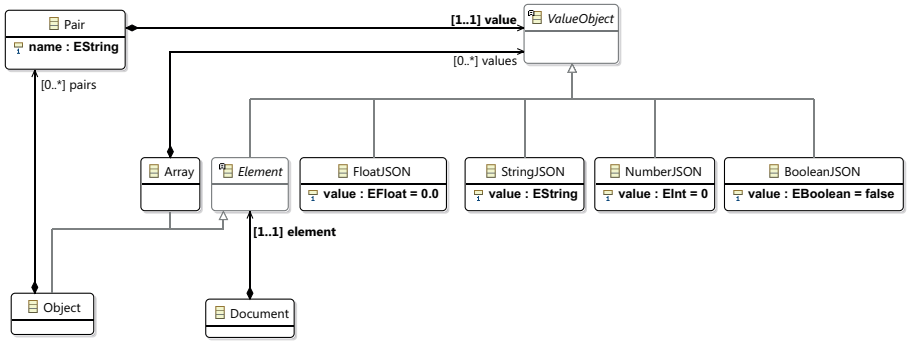


Figura 4. Metamodelo *JSON* definido a partir de la gramática de JSON.

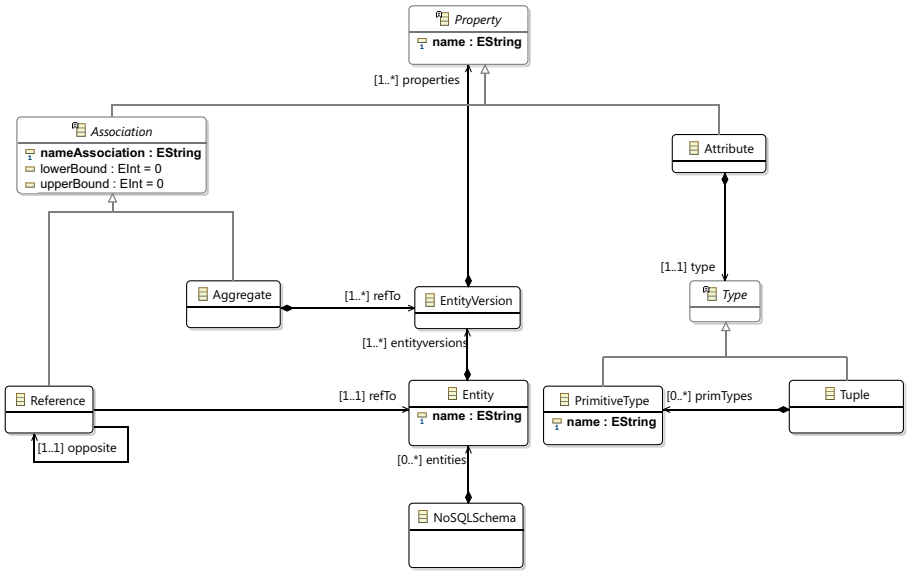


Figura 5. Metamodelo *NoSQL-Schema* que representa esquemas de datos NoSQL.

calcular sus tipos. Por ejemplo, en el código de la Figura 2, los tipos de los primeros objetos de las colecciones *Publisher* y *Book* serían respectivamente (no se considera el campo *id*): *<name:String, city:tuple(String)>* y *<title:String, publisher_id:String, year:Integer, authors:tuple(Author), content:Content>*. En realidad *Author* denota al tipo *<name:String, company:<name:String, city:String)>>* para el primer autor embebido y *Content* denota al tipo *<pages:Integer, chapters:Integer>*.

Entidades. Se crea una versión de una entidad (*EntityVersion*) por cada tipo de objeto JSON del modelo de entrada, ya sea raíz o esté embebido en

otro objeto. Cuando se encuentra la primera versión de un tipo de objeto, también se crea la entidad (*Entity*) que se asocia a la primera versión. Por ejemplo, en el caso del código de la Figura 2, se generaría una entidad *Book* que tendría dos versiones que corresponderían a los tipos <*title:String, publisher_id:String, year:Integer, authors:tuple(Author), content:Content*> y <*title:String, authors:tuple(Author), published_date:Date, pages:Integer*> y también se generarían dos versiones para *Author* que corresponderían a los tipos <*name:String, company:Company*> y <*name:String, country:String*>. El tipo de *Company* sería <*name:String, city:String*> y el de *Date* sería <*date:String*>. Cuando se recorren los objetos JSON, para cada objeto se obtiene su tipo y se compara con el resto de tipos para ver si es necesario generar una nueva versión de una entidad (*EntityVersion*). Obsérvese que un modelo NoSQL Schema no registra el tipo de cada objeto JSON sino sólo las versiones.

La obtención del nombre de una entidad raíz es simple en el caso de MongoDB, ya que las colecciones tienen un nombre. Sin embargo, para otros sistemas se analizarán las convenciones propuestas. Por ejemplo, en CouchDB existe la convención de que cada objeto incluya una propiedad *type* cuyo valor es el nombre del tipo que denota la entidad. Si no se puede deducir, el nombre debe ser proporcionado por el usuario. En el caso de un objeto embebido, es la clave del par, pero pasando el nombre de plural a singular si es necesario.

Atributos. Para cada versión de entidad inferida, se genera un atributo (*Attribute*) por cada par JSON (*Pair*) del correspondiente objeto JSON cuyo valor sea o bien un tipo primitivo o bien un *array* cuyos valores sean todos de tipo primitivo. En el primer caso se liga el atributo a un tipo (*PrimitiveType*) y en el segundo a un tipo (*Tuple*). El nombre del atributo es el nombre del par. El tipo primitivo *Date* se registra de modo diferente en cada sistema, normalmente como un *string* con una determinada estructura (en el ejemplo se puede ver para MongoDB).

Relación de agregación. Para cada versión de una entidad inferida, se genera una relación de agregación (*Agreggate*) por cada par JSON (*Pair*) cuyo valor sea un elemento embebido (objeto o *array* de objetos). La cardinalidad de la relación será uno a uno si el elemento embebido es un objeto, o uno a muchos si es un *array* de objetos. El nombre de la relación coincide con el de par JSON aunque hay algunas excepciones. Por ejemplo, el nombre se pasa de singular a plural si la cardinalidad es de cero o uno a muchos. La asociación *refTo* entre *Agregate* y *EntityVersion* se resuelve una vez se han creado todas las entidades y sus versiones, y registra todas las versiones de entidad que corresponden a una entidad anidada. En el ejemplo, *Book* agrega dos versiones de la entidad *Author*.

Referencias. Las referencias (*Reference*) se deducen de dos formas:

- Primero se comprueban convenciones para expresar referencias como:
  - Un par (*Pair*) con nombre *nameEntity_id* indica que existe una referencia a la entidad de nombre *nameEntity*.

- (MongoDB) Un objeto {*_ref: "collection_name", _id: "reference_id"*} indica una referencia a la entidad *collection_name*.

- Si existe un par cuyo nombre corresponde (*matches*) con el nombre de una entidad existente y su valor es un entero o un *array* de enteros que son identificadores de esa entidad (campo *_id*).
  Una *Reference* se asocia a la *Entity* referenciada (*refTo*). Por ejemplo, *Book* referencia a *Publisher* en nuestro ejemplo. Tanto *refTo* y las cardinalidades se resuelven una vez se han inferido todas las entidades y referencias. La referencia reflexiva *opposite* se resuelve una vez se han generado todas las referencias, para lo que se comprueba si la entidad a la cual se hace referencia contiene un *Reference* que haga referencia a la entidad origen.

La Figura 6 muestra a la derecha la representación gráfica generada del esquema inferido para el ejemplo de la Figura 2. Se trata de un diagrama de clases UML y se puede observar que se han inferido las entidades raíz *Book*, *Publisher* y *Journal*, así como las entidades *Author* y *Content*, agregadas a *Book*, y la entidad *Company*, agregada a *Author*. También se muestran las referencias entre *Book* y *Publisher*, y entre *Publihser* y *Journal*. Este tipo de representación no permite mostrar las diferentes versiones de una entidad, aunque sí que existen varias versiones de una propiedad, por ejemplo se observa que los autores se registran como una tupla de *strings* o un agregado. Por otra parte, a la izquierda de la Figura 6 se muestra el informe textual generado que sí especifica todas las versiones de una entidad indicando sus propiedades (nombre y tipo).

5. Implementación

El trabajo se ha desarrollado sobre la plataforma *Eclipse Modeling Framework* (EMF) [8]. El inyector de modelos JSON se ha obtenido con la herramienta EMFText[6] (*p*lugin de Eclipse) que permite definir DSL textuales. Dado el metamodelo Ecore del lenguaje, EMFText ofrece una notación para expresar la gramática del DSL y genera el inyector de modelos y un editor. En nuestro caso, la gramática JSON es muy pequeña y el inyector se ha obtenido fácilmente.

La transformación m2m que implementa el proceso de inferencia se creó con el lenguaje RubyTL [6], que es un DSL embebido en Ruby que combina reglas declarativas con la posibilidad de escribir código Ruby. Este lenguaje proporciona un mecanismo para organizar las transformaciones en módulos (se denominan *fases*) que se pueden componer a través de un tipo especial de regla que permite refinar los elementos del modelo generados en fases ejecutadas anteriormente [6]. Este mecanismo se ha aprovechado para organizar nuestra transformación en cuatro fases. En la primera fase se crean las entidades y sus diferentes versiones, y las propiedades. En la segunda fase se crean referencias para las propiedades cuyo tipo se identificó como una tupla de enteros en la primera fase y también se generan los tipos *Date* que en la primera fase son registrados como agregados.

[6] `http://www.emftext.org/index.php/EMFText`.

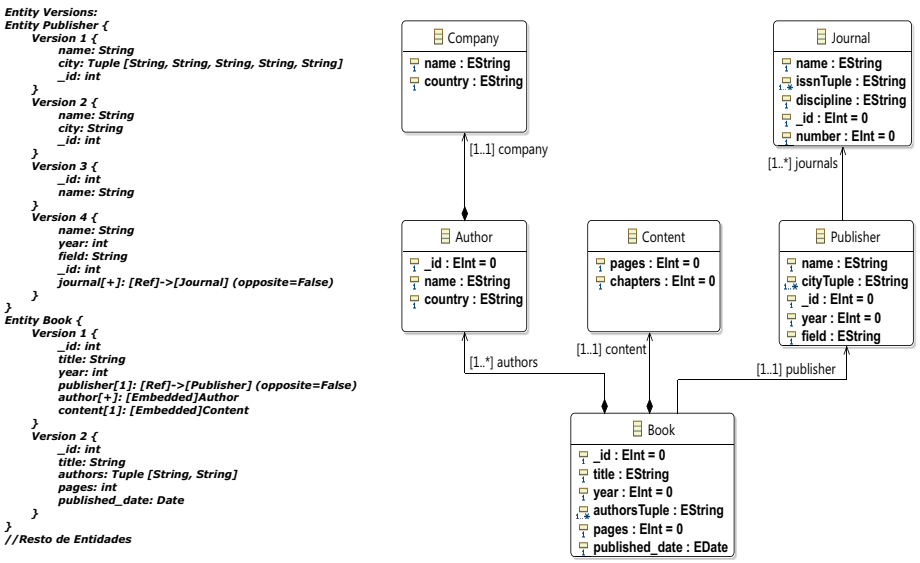


Figura 6. Esquema inferido para los datos JSON de la Figura 2.

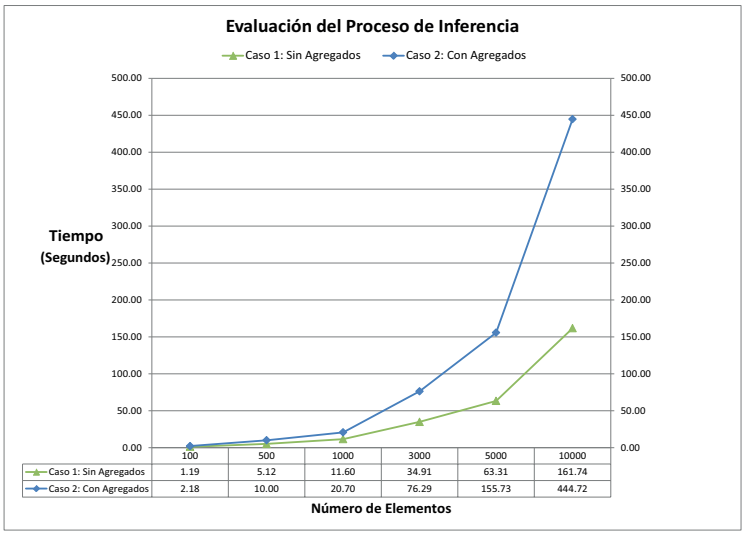
En la siguiente fase se resuelve la referencia *refTo* para *Reference* y *Aggregate* así como las cardinalidades, y en la última fase, una vez que se tienen generadas todas las referencias, éstas son analizadas para resolver la referencia *opposite*.

La transformación m2t que genera el informe textual que especifica las versiones de las entidades se ha implementado con el lenguaje MOFScript[7]. También se ha implementado en este lenguaje la transformación m2t que genera código EMFatic para expresar los modelos NoSQL Schema como metamodelos Ecore. Esta transformación establece la siguiente correspondencia: i) metaclases (*EClass*) con entidades, ii) atributos de una metaclase (*EAtrribute*) con atributos de una entidad, iii) referencias contenedoras (*EReference con atributo composite=true*) con relaciones de agregación entre entidades, y iv) referencias no contenedoras (*EReference con atributo composite=false*) con referencias entre entidades. De este modo, un esquema se muestra como un diagrama de clases por medio de editores, en concreto se ha utilizado Sirius[8]. En la Figura 6 se puede observar el diagrama generado por Sirius para el esquema del ejemplo.

La herramienta que hemos construido puede ser descargada de `http://www.catedrasaes.org/wiki/NoSQLSchemaVersions`.

Se han medido los tiempos de ejecución de todas las etapas del proceso para dos casos de pruebas, uno con una única versión de *Book* que no contiene agregados y que referencia a una única versión de *Publisher* y otra con cuatro versiones

[7] `http://eclipse.org/gmt/mofscript/`.
[8] `https://www.eclipse.org/proposals/modeling.sirius/`.



	100	500	1000	3000	5000	10000
Caso 1: Sin Agregados	1.19	5.12	11.60	34.91	63.31	161.74
Caso 2: Con Agregados	2.18	10.00	20.70	76.29	155.73	444.72

Figura 7. Tiempos de proceso con diferentes configuraciones.

de *Book* en las que una de ellas agrega a autores que a su vez agregan empresas, y todas ellas referencias a una versión de *Publisher*. Ambos casos de prueba se han ejecutado para 100, 500, 1.000, 3.000 y 10.000 objetos *Book*. En el segundo caso, la mitad de los objetos *Book* correspondían con la versión que agrega a autores y la otra a versiones que sólo incluyen propiedades con valores de tipo primitivo y que se diferencian en que el campo *authors* es una tupla de *strings*, un valor *string*, o bien no se incluye. Como se observa en la Figura 7 el tiempo de ejecución de la transformación m2m crece exponencialmente con el número de objetos en ambos casos y la existencia de objetos embebidos incrementa el coste en un factor que crece de 1,83 a 2,75 al pasar de 100 a 10.000 elementos. No se han considerado los tiempos de ejecución de los procesos de inyección y generación, ya que su coste está por debajo de 1 y 5 segundos, respectivamente, en el peor caso considerado.

6. Trabajo relacionado

Dado que la iniciativa JSON Schema[9] es reciente y su adopción es limitada, la obtención de esquemas a partir de datos JSON es un área de trabajo muy activa, dado que JSON es ampliamente utilizado y muchas herramientas y aplicaciones podrían beneficiarse. Por ejemplo, en el caso de escenarios de integración de datos manejados por servicios basados en JSON, los desarrolladores deben conocer la estructura de esos datos. Para este escenario, se ha presentado una herramienta

[9] `http://json-schema.org/`.

(JSON discoverer[10]) que extrae el esquema implícito en servicios basados en JSON y que sin duda es el trabajo más relacionado con nuestra propuesta.

JSON Discoverer infiere el esquema en tres etapas [1]. Primero se inyectan los datos JSON de un servicio en un modelo que conforma a un metamodelo similar al nuestro (la principal diferencia es que ellos distinguen objetos agregados de los objetos raíz y nosotros los tratamos de forma homogénea). En la segunda etapa, se establece una correspondencia entre el modelo JSON el metamodelo Ecore para obtener un modelo del dominio, y finalmente se combinan los modelos de cada servicio considerando las clases comunes. Nuestra estrategia de inferencia del esquema es distinta dado que nosotros manejamos documentos JSON que no corresponden a servicios sino a datos NoSQL y, por tanto, nuestro propósito es inferir el esquema de bases de datos NoSQL en vez de tratar servicios web. La diferencia más significativa es que JSON Discoverer no maneja entidades con diferentes versiones ni infiere referencias entre entidades, como se puede comprobar al ejecutar la herramienta en su página web con nuestro ejemplo. Finalmente, señalar que nosotros también hemos aplicado un *mapping* a modelos Ecore, pero no de los modelos JSON, sino de los modelos del esquema inferido. Además, se ha implementado mediante una transformación m2t en vez de una m2m ya que se pretendía generar un documento EMFatic para obtener de forma automática una representación gráfica del esquema.

Recientemente se ha anunciado Drill[11], que genera y ejecuta consultas SQL para sistemas NoSQL. En la documentación se puede leer que Drill descubre el esquema dinámicamente durante el procesamiento, pero no se dispone de una herramienta separada que proporcione tal facilidad y genere representaciones del esquema para los desarrolladores de aplicaciones que manejan sistemas NoSQL ni se conoce la estrategia aplicada. Además, por la estructura de tabla similar a SQL, Drill no es capaz de acomodar entidades con diferentes columnas.

Hay algunas herramientas que establecen una correspondencia (serialización e inyección) entre modelos Ecore y documentos JSON, bien en los dos sentidos como *emfjson*[12] o en uno sólo como *xmi-to-json*[13]. Estas herramientas no realizan ningún tipo de inferencia sino que sólo cubrirían la fase inicial de inyección de nuestro proceso.

7. Conclusiones y trabajo futuro

En este trabajo se ha presentado una solución MDE para inferir el esquema de bases de datos NoSQL. La principal novedad con respecto a otras propuestas de descubrimiento del esquema implícito en datos JSON es la identificación de todas la versiones de los objetos, lo cual es importante, dado que los datos NoSQL son no uniformes. Otra característica importante de la solución propuesta es la independencia del sistema NoSQL y de los diferentes formatos JSON que

[10] `http://atlanmod.github.io/json-discoverer`.
[11] `http://drill.apache.org/`.
[12] `http://emfjson.org/`.
[13] `https://github.com/dsevilla/xmi-to-json`.

usan para representar sus datos. Se ha implementado una primera versión de la herramienta, que se puede mejorar en varios aspectos: i) realizar optimizaciones para mejorar la eficiencia de la transformación m2m que implementa el proceso de inferencia del esquema, ii) definir un tipo de diagrama que muestre claramente las diferentes versiones de los objetos y otro diagrama que muestre los objetos similar al utilizado en JSON Discoverer, iii) crear un servicio web que permita a los usuarios introducir el código JSON o un enlace a los datos y visualice los informes textuales y diagramas que especifican el esquema inferido, iv) proporcionar un API que permita usar nuestra solución por los desarrolladores. Por otra parte, la inferencia del esquema será aprovechada para generar validadores y capas de acceso a bases de datos NoSQL destinados asegurar que las aplicaciones NoSQL acceden correctamente a los datos [3]. Otro trabajo futuro investigará la ejecución distribuida de la transformación m2m considerando que los datos se almacenan distribuidos en *clústers*.

Referencias

1. J.L. Cánovas Izquierdo, Jordi Cabot: Discovering Implicit Schemas in JSON Data. In ICWE 2013: 68-83.
2. Eric Evans: Domain-Driven Design. Tackling Complexity in the Heart of Software. Addison-Wesley, 2006.
3. Martin Fowler: Schemaless Data Structures; enero, 2013 (`http://martinfowler.com/articles/schemaless/`)
4. Eric Redmond, Jim R. Wilson: Seven Databases in Seven Weeks. A Guide to Modern Databases and the NoSQL Movement. Pragmatic Programmers, 2013.
5. P.J. Sadalage, Martin Fowler: NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, 2012.
6. J. Sánchez Cuadrado, J. García Molina: Modularization of Model Transformations Through a Phasing Mechanism. Software and System Modeling 8(3): 325-345 (2009).
7. O. Sánchez Ramón, F.J. Bermúdez Ruiz, Jesús García Molina: Una valoración de la Modernización de Software Dirigida por Modelos. En JISBD 2013, 288-302, Madrid (Spain).
8. Dave Steinberg, F. Budinsky, M. Paternostro, Ed Merks: Eclipse Modeling Framework, 2nd Edition, Addison-Wesley, 2008.