UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

Inferring NoSQL Data Schemas with Model-Driven Engineering Techniques

Inferencia de Esquemas en Bases de Datos NoSQL por medio de Ingeniería Dirigida por Modelos

D. Severino Feliciano Morales

2017

Inferring NoSQL Data Schemas with Model-Driven Engineering Techniques

A DISSERTATION PRESENTED
BY
SEVERINO FELICIANO MORALES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

SUPERVISED BY
JESÚS GARCÍA MOLINA
DIEGO SEVILLA RUIZ

UNIVERSITY OF MURCIA
MURCIA, SPAIN
MARCH 2017

Inferencia de esquemas en Bases de Datos NoSQL por medio de Ingeniería Dirigida por Modelos

Resumen en Español

Los requisitos de la Web 2.0 y las nuevas tecnologías que han aparecido recientemente (por ejemplo, Internet de las Cosas, aplicaciones móviles o Big Data) han evidenciado las limitaciones de los sistemas de gestión de bases de datos (SGBD) relacionales cuando se usan en las modernas aplicaciones que requieren manejar grandes volúmenes de datos. Esto ha motivado el desarrollo de un número cada vez mayor de sistemas no relacionales con el propósito de abordar los requisitos de tales aplicaciones, en especial, la capacidad de representar datos complejos, la escalabilidad y hacer frente al aumento del tráfico de datos. El término NoSQL (*Non SQL/Not only SQL*) se utiliza para denotar esta nueva generación de sistemas de bases de datos.

El interés en los sistemas NoSQL ha crecido constantemente durante la última década. Un gran número de empresas ya han implantado bases de datos NoSQL y la adopción aumentará considerablemente en los próximos años, como se informa en [1, 87]. El sitio web "nosql-database.org" muestra una lista de unos 225 sistemas NoSQL disponibles en la actualidad. En realidad, el término NoSQL se refiere a un variado conjunto de paradigmas de modelado de datos que gestionan datos semi-estructurados y no estructurados. Las principales categorías de NoSQL son: *documentos*, *familia de columnas*, *clave-valor* y *bases de datos de grafos*. Las tres primeras representan datos semi-estructurados utilizando un modelo de datos basado en la agregación [109]. Estos paradigmas son los más extendidos, siendo MongoDB [80] el sistema NoSQL más utilizado [88].

Es preciso señalar que los sistemas NoSQL que pertenecen al mismo paradigma pueden tener diferentes características. Sin embargo, la mayoría de los sistemas NoSQL tienen algunas propiedades comunes, como son: SQL no es utilizado, no tiene que ser definido un esquema para especificar la estructura de datos, la ejecución en clústeres es el factor principal que determina su diseño, y se desarrollan como iniciativas *open-source* [109]. Poder almacenar datos sin tener que definir previamente un esquema es uno de las características más atractivas de las bases de datos NoSQL, aunque conlleva importantes desventajas como señalaremos más adelante en este capítulo.

El reciente informe de Dataversity "Insights on Modeling NoSQL" [1] ha señalado que el modelado de datos también será una actividad crucial para las bases de datos NoSQL y ha llamado la atención sobre la necesidad de herramientas NoSQL que proporcionen funcionalidad similar a la disponible para bases de datos relacionales. Los autores de este informe

identificaron tres principales características que deberían ser ofrecidas por las herramientas de modelado NoSQL: visualización de modelos, generación de código y gestión de metadatos.

La *Ingeniería de Datos* es la disciplina que se ocupa de los principios, técnicas, métodos y herramientas relacionado con la gestión de datos en el desarrollo de software. Los datos se almacenan normalmente en sistemas de gestión de bases de datos (por ejemplo, relacionales, orientados a objetos o NoSQL) y la Ingeniería de Datos se ha ocupado principalmente de las bases de datos relacionales hasta el momento, aunque el interés se está desplazando hacia las bases de datos NoSQL.

En esta tesis, hemos abordado cuestiones relacionadas a la Ingeniería de Datos, más específicamente con la ingeniería inversa de datos NoSQL y el desarrollo de utilidades de bases de datos capaces de proporcionar los tres tipos de funcionalidad mencionados anteriormente. El trabajo de esta tesis se ha centrado, por lo tanto, en la Ingeniería de Datos NoSQL que es un área de investigación emergente dentro del campo de Ingeniería de Datos.

En los últimos años, *la Ingeniería de Software Dirigida por Modelos* (MDSE o simplemente MDE) está ganando cada vez más aceptación, principalmente debido a su capacidad para abordar la complejidad del software y mejorar la productividad del software [110, 121, 16]. MDE promueve el uso sistemático de modelos con el fin de elevar el nivel de abstracción en el que se especifica el software, y para aumentar el nivel de automatización en el desarrollo de software. Las técnicas MDE, en especial el metamodelado y las transformaciones de modelos, han demostrado ser útiles para tareas de ingeniería directa e inversa.

Los esquemas de datos son modelos y las operaciones sobre ellos se pueden implementar usando transformaciones de modelo. Por lo tanto, los enfoques transformacionales se han utilizado tradicionalmente para automatizar tareas de ingeniería de datos como normalización, conversión de esquemas o integración de esquemas como se explica en detalle en [56]. La implementación de estas tareas podría ser facilitada por MDE. Sin embargo, la comunidad de ingeniería de datos ha prestado poca atención a la aplicación de MDE como se indica en [105].

Motivación A diferencia de los sistemas relacionales, en la mayoría de las bases de datos NoSQL los datos se almacenan sin necesidad de haber definido previamente un esquema. Esta falta de un esquema de datos explícito (*schemaless*) es probablemente la característica NoSQL más atractiva para los desarrolladores de bases de datos. Siendo *schemaless*, se proporciona una mayor flexibilidad para manejar los datos, por ejemplo, la base de datos puede almacenar datos con una estructura diferente para el mismo tipo de entidad (datos no uniformes) y la evolución de los datos es favorecida debido a la falta de restricciones impuestas a la estructura de datos.

Sin embargo, eliminar la necesidad de declarar esquemas explícitos no tiene que confundirse con la ausencia de un esquema, ya que éste está implícito en los datos almacenados y en el código de las aplicación. De hecho, los desarrolladores siempre deben tener en cuenta

el esquema cuando escriben código que accede a la base de datos. Por ejemplo, tienen que respetar los nombres y tipos de los campos al escribir operaciones de inserción o consulta. Esta es una tarea propensa a errores, más aún cuando la existencia de varias versiones de cada entidad es probable. Cuando existe un esquema es posible un control estático de errores. Por otro lado, algunas herramientas y utilidades de base de datos necesitan conocer el esquema para ofrecer funcionalidades tales como realizar consultas tipo SQL, migrar datos automáticamente o bien procesar datos de manera eficiente.

Por tanto, está surgiendo un interés creciente en el manejo de esquemas NoSQL explícitos, como se evidencia en los enfoques de inferencia de esquemas recientemente propuestos que serán analizados en esta tesis [72, 119, 102], y algunas herramientas ya disponibles para ayudar a los desarrolladores de NoSQL [81, 108, 47]. En cuanto a la dificultad de escribir código que acceda correctamente a los datos, aparecen dos estrategias para las aplicaciones NoSQL: (i) combinar la carencia de un esquema con mecanismos que garanticen un acceso correcto a los datos (por ejemplo, validadores de datos) y (ii) utilizar mappers que convierten datos NoSQL en objetos de un lenguaje de programación. La mayoría de los mappers actuales son para sistemas NoSQL de documentos (Object-Document mappers, ODM), y Mongoose [81] es el ODM más utilizado, que ha sido creado para usar MongoDB con Javascript. Sin embargo, ODM para otros lenguajes, como Java y PHP, también están disponibles.

Como se ha indicado anteriormente, la ausencia de un esquema permite almacenar datos no uniformes, es decir, una entidad puede tener versiones diferentes del esquema implícito (tipo) que caracteriza los datos almacenados de dicha entidad. Cada versión está definida por un conjunto de atributos. La versión de una entidad puede originarse, por ejemplo, mediante una elección de diseño inicial o cambios en el esquema implícito de una entidad (los objetos que tienen el esquema evolucionado coexisten con los objetos que tienen el nuevo esquema). Por lo tanto, teniendo en cuenta que cada entidad puede tener una o más versiones, establecer el esquema (o tipo) de una entidad requiere considerar el conjunto de esquemas de sus versiones. Por lo general, la unión de todas las versiones de entidad [72] o alguna forma de tipo aproximado [119] se considera el esquema de una entidad. Una noción de esquema de base de datos global para bases de datos NoSQL no se ha propuesto todavía hasta donde sabemos.

El informe "Insights into Modeling NoSQL" [1], presentó recientemente un análisis sobre el papel del modelado para los sistemas NoSQL. Sus autores destacaron que las herramientas de modelado de datos son esenciales para abordar algunos de los principales retos para la adopción industrial de los sistemas NoSQL como: modelado de datos, gobernabilidad de los datos, documentación y herramientas apropiadas. Hay una carencia de herramientas de bases de datos NoSQL que proporcionen capacidades similares a las que se disponen para bases de datos relacionales. Además, las herramientas existentes son inmaduras. El informe identificó tres categorías principales de funcionalidades deseadas para los sistemas NoSQL: *visualizar diagramas de esquemas y datos*, *generación de código* y *manejo de metadatos*.

La visualización de modelos, la documentación y la ingeniería inversa de la base de datos NoSQL existente se señalan como características que se proporcionan mediante herramientas de visualización. Los diagramas serían útiles para diseñar bases de datos, tomar decisiones, documentar y representar los esquemas inferidos mediante procesos de ingeniería inversa. El informe señaló que las bases de datos NoSQL son *schemaless*, por lo que se necesita un proceso de ingeniería inversa para descubrir el esquema implícito en los datos y código. En cuanto a la generación de código, los modelos podrían utilizarse para generar artefactos de aplicaciones, por ejemplo, código de una aplicación podría generarse a partir de esquemas NoSQL de alto nivel o esquemas físicos podrían generarse a partir de esquemas conceptuales o lógicos. Esta generación de código sería útil para aumentar la productividad y mejorar la calidad (por ejemplo, el número de errores se reduce), entre otros beneficios. Finalmente, los metadatos se consideran esenciales en algunos escenarios como lograr la integración entre diferentes almacenes NoSQL. Cabe señalar que la información expresada en los esquemas es la parte esencial de los metadatos gestionados en las bases de datos. Los autores del informe señalaron que la persistencia poliglota será la norma en un futuro muy cercano. Esto exigirá la integración de sistemas de bases de datos de diferentes tecnologías. Los modelos podrían ser útiles para tal integración, y las herramientas de la base de datos tendrán que ser capaces de manejar modelos para diferentes paradigmas.

Cuando los objetivos de esta tesis se establecieron a principios de 2014, todavía no se habían publicado trabajos sobre la inferencia de esquemas para bases de datos NoSQL, y no habían herramientas NoSQL disponibles. Por ejemplo, el artículo que describe nuestra primera versión del proceso de inferencia fue presentado en CIbSE'2105 en diciembre de 2014 [82], y la segunda versión fue descrita en un artículo que enviamos a ER'2015 en abril de 2015 [102]. Estos trabajos ya presentaron algunas soluciones para la visualización de esquemas y generación de código para mappers ODM. Los primeros enfoques de inferencia de esquemas propuestos por otros grupos de investigación también se publicaron a lo largo del año 2015, así como las primeras herramientas de visualización de esquema NoSQL [47] y los mappers ODM [81].

Definición del problema y Objetivos La ausencia de esquema explícito es una característica necesaria en los sistemas NoSQL debido a que proporciona la flexibilidad requerida por el hecho de que la estructura de datos puede cambiar con frecuencia. Sin embargo, los desarrolladores necesitan comprender el esquema implícito cuando escriben código y algunas herramientas requieren conocer el esquema para soportar alguna funcionalidad. Esto exige la definición de estrategias de ingeniería inversa dirigidas a descubrir los esquemas y el desarrollo de herramientas que aprovechen los esquemas inferidos para ofrecer funcionalidades que ayuden a los desarrolladores. Estas herramientas permitirán que la flexibilidad obtenida no suponga perder los beneficios importantes que proporcionan los esquemas de base de datos. La existencia de versiones de entidad en bases de datos NoSQL es el principal reto que se debe abordar en el proceso de inferencia de esquemas. Con respecto a la implementación

del proceso de inferencia y las herramientas de la base de datos, creemos que las técnicas de MDE, especialmente el metamodelado y las transformaciones de modelos, facilitan esta tarea. El metamodelado es útil para tener representaciones en alto nivel de abstracción de la información manejada, y las transformaciones del modelo ayudan a automatizar el desarrollo. El trabajo de esta tesis se centrará en los sistemas NoSQL cuyos modelos de datos están orientados a la agregación.

Los objetivos de esta tesis son los siguientes:

Objetivo 1. Diseño e implementación de un proceso de inferencia de esquemas. Diseñar un enfoque de ingeniería inversa basado en modelos para inferir los esquemas implícitos en bases de datos NoSQL. La estrategia definida debe tener en cuenta la existencia de versiones de entidades y obtener un modelo que represente todas las versiones existentes en la base de datos para cada entidad.

Objetivo 2. Definición de una noción de esquema de bases de datos NoSQL Realizar investigaciones sobre cómo el concepto tradicional de esquema conceptual de base de datos relacional puede trasladarse a bases de datos NoSQL. Una definición que incluya entidades o versiones de entidad y relaciones entre entidades, en particular agregación y referencia.

Objetivo 3. Diseñar diagramas para representar esquemas NoSQL y construir herramientas que soporten su visualización. Investigar qué tipo de diagramas podrían representar visualmente los diferentes tipos de esquemas identificados para las bases de datos NoSQL. También debemos implementar algunas herramientas que soporten la visualización de estos diagramas. Las técnicas MDE se utilizarían para implementar dichas herramientas.

Objetivo 4. Generación de código para mappers ODM. Desarrollar una solución MDE para generar automáticamente código para mappers ODM, como esquemas y funciones de validación para Mongoose y otros ODM.

Objetivo 5. Generación de validadores de datos. Desarrollar una solución MDE para generar automáticamente validadores (es decir, predicados de esquema) destinados a ser utilizados para comprobar que los datos se almacenan sin violar el esquema.

Objetivo 6. Definición de una estrategia para clasificar los objetos. Definir un algoritmo de agrupación para determinar a qué versión de entidad pertenece un dato leído de la base de datos. Esta clasificación podría ser útil como un primero paso para una implementación de migración automática de objetos de una versión a otra, o para homogeneizar bases de datos.

Metodología Para la consecución de los objetivos de esta tesis se ha seguido la metodología "Design science research" (DSRM) descrita en [68, 117]. Define un proceso que consiste de seis actividades: (1) Identificación del problema y motivación, (2) Definir los objetivos, (3) Diseño y desarrollo de la solución, (4) Demostración, (5) Evaluación y (6) Conclusiones y difusión. Se trata de un proceso iterativo en el que el conocimiento producido a lo largo del proceso, por medio de la construcción y evaluación de nuevos artefactos, sirve como realimentación que permite mejora los artefactos creados con anterioridad hasta completar la solución final.

Discusión de los resultados Hemos definido un enfoque de inferencia de esquemas que se ha implementado como un proceso de ingeniería inversa basado en modelos. El esquema deducido se representa como un modelo que conforma al metamodelo NoSQL_Schema que es independiente de la plataforma. Las principales diferencias de nuestra estrategia de inferencia con respecto a otros enfoques propuestos son (i) se extraen las versiones de cada entidad; (ii) se descubren todas las relaciones entre las versiones de entidad extraídas: agregación y referencias; (iii) se considera la escalabilidad y el rendimiento del algoritmo de inferencia aplicando una operación Map-Reduce para acceder directamente a la base de datos y obtener el conjunto mínimo de objetos JSON necesarios para aplicar el proceso de inferencia. Nuestro interés no es obtener un esquema simplificado, aproximado o esqueleto, sino que nuestra idea es registrar todas las versiones de la entidad y las relaciones entre ellas. Esta decisión está motivada por el hecho de que nuestro enfoque está dirigido a aplicaciones de negocio en las que el número máximo de versiones de una entidad no será excesivamente grande. Este escenario es diferente al considerado en [119] que supone que varias decenas de miles de versiones pueden existir para una entidad. El enfoque ha sido validado por medio de una base de datos MongoDB generada a partir de los datos de StackOverflow. Por lo que sabemos, nuestro trabajo es el primer enfoque que maneja esquemas versionados. Otros enfoques no toman en cuenta la versión de las entidades [23] o bien obtienen el esquema de unión [72] o un esquema aproximado [119]. Aquí, hemos definido la noción de esquema versionado y los siguientes tipos: *Esquema de version* (raíz o agregado) que sólo incluye las versiones de entidades relacionadas con una versión de entidad (y entidades si hay referencias); *Esquema entidad* (raíz o agregado) que sólo incluye entidades relacionadas con una entidad; *Esquema de base de datos* que incluye todas las versiones de entidad de una base de datos; y *Esquema de entidades de la base de datos* que incluye los esquemas unión de todas las entidades de la base de datos. Por lo tanto, hemos identificado un conjunto de esquemas para sistemas NoSQL, que pueden definirse en tres niveles: entidad, versión de entidad o base de datos.

Los desarrolladores de aplicaciones de base de datos NoSQL necesitan entender el esquema implícito de la base de datos. De hecho, deben tener en cuenta este esquema cuando escriben o mantienen código. La visualización del esquema en forma de diagramas sería muy útil para estos desarrolladores de la misma manera que los esquemas E/R se han utilizado

para los desarrolladores de aplicaciones de bases de datos relacionales. Hemos definido representaciones visuales para cada tipo de esquema definido en esta tesis. En particular, hemos utilizado diagramas de clases UML para representar esquemas de versión raíz, esquemas de unión de entidades y esquemas de bases de datos de entidades. Se obtienen varios beneficios al representar esquemas NoSQL en forma de diagramas: se facilita su comprensión, su comunicación y se obtiene una documentación separada del código.

Hemos desarrollado dos soluciones MDE para visualizar los tipos de diagramas definidos. En primer lugar, transformamos el modelo de esquema generado en el proceso de inferencia en un metamodelo Ecore con el objetivo de visualizarlo mediante un editor de metamodelos Ecore. Hemos utilizado el editor integrado en Eclipse/EMF. Esta solución ha ilustrado los beneficios de representar modelos y metamodelos uniformemente. Después de esta prueba de concepto, transformamos el modelo de esquema inferido en código PlantUML para una visualización del diagrama de clases UML. Nuestro trabajo ha servido para definir una notación específica para los esquemas NoSQL [29].

Con el objetivo de ilustrar posibles aplicaciones de los esquemas inferidos, además de su visualización, hemos abordado el desarrollo de dos utilidades de generación de código: código para mappers ODM y código para validación de datos. Hemos diseñado e implementado un enfoque para la generación de código para mappers ODM existentes. Es una solución independiente de tecnología, y como prueba de concepto se ha aplicado a Mongoose, donde hemos sido capaces de generar esquemas y artefactos para diferentes funcionalidades proporcionadas por este mapper, como validadores, discriminadores y manejo de referencias. La solución MDE ideada ha mostrado la utilidad de definir metamodelos intermedios en una cadena de transformaciones de modelos. Hemos definido un metamodelo que reorganiza la información incluida en un modelo de esquema en una forma más apropiada para generar los esquemas de cada versión, tarea que requiere distinguir entre propiedades comunes y específicas para cada versión. La validación de datos es necesaria para asegurar que todos los objetos recuperados y almacenados por una aplicación se ajusten a una versión de entidad dada.

Hemos desarrollado una solución MDE que genera validadores para ser aplicados cuando los datos se almacenan en la base de datos. Se trata de una cadena de transformación de modelos de dos pasos. Aquí, se ha definido también un metamodelo para una representación intermedia que facilita la generación de validadores. Los modelos intermedios se obtienen mediante una transformación modelo a modelo a partir del modelo del esquema y, a continuación, una transformación modelo a texto genera las funciones de validación que comprueban la versión de entidad de los datos que se van a almacenar.

Por otro lado, se ha diseñado e implementado una utilidad que clasifica los datos en versiones de entidad. Esta solución ha implicado abordar el problema de determinar a qué versión de entidad pertenece un objeto recuperado de la base de datos. Para ello, se ha diseñado un algoritmo de *clustering* que se basa en un algoritmo de generación de árboles de decisión óptimos.

Cabe señalar que un objetivo general de esta tesis fue mostrar los beneficios de usar MDE en el área emergente de Ingeniería de Datos NoSQL. Hemos creado soluciones MDE tanto para implementar el proceso de inferencia de esquema como las utilidades de base de datos desarrolladas para ilustrar posibles aplicaciones de los modelos de esquema inferidos. Estas soluciones han mostrado algunas de las principales ventajas que ofrece MDE como (i) la capacidad del metamodelo para representar información a alto nivel de abstracción, la cual ha sido especialmente útil en el proceso de ingeniería inversa y para obtener representaciones intermedias en las cadenas de transformación de modelos definidas, (ii) el poder unificador de los modelos, (iii) la utilidad de las transformaciones de modelos para generar artefactos como código de las aplicaciones, (iv) aprovechar la existencia de las herramientas existentes para la definición de DSL textuales y gráficos, y (v) la utilidad de los modelos para conseguir independencia de la plataforma.

Contribuciones Las principales contribuciones de esta tesis son las siguientes. Hemos definido el primer enfoque que infiere esquemas conceptuales de bases de datos NoSQL que tiene en cuenta las versiones de las entidades y sus relaciones así como la escalabilidad. El metamodelo de esquemas contribuye a la propuesta de modelos unificados para bases de datos NoSQL. Nuestro análisis de la noción de esquema para bases de datos agregadas NoSQL es otra contribución de esta tesis. El conjunto de esquemas propuestos podría servir para guiar el trabajo de investigación en este dominio. Hasta donde sabemos, hemos diseñado los primeros diagramas para visualizar esquemas versionados NoSQL, tanto la propuesta de usar diagramas de clase UML como la definición de una notación específica. Hemos diseñado e implementado las primeras utilidades para la generación de código para mappers ODM. En particular, hemos generado código para el *mappers* Mongoose, aunque la solución MDE definida es aplicable a cualquier *mapper* ODM. También son originales las soluciones propuestas para la generación de validadores de datos y clasificación de datos en versiones de entidad. El desarrollo de estas herramientas también ha contribuido a mostrar cómo las técnicas MDE pueden ser muy útiles en el área de Ingeniería de Datos NoSQL. Aunque la aplicación de MDE en Ingeniería de Datos ha sido muy limitada hasta la fecha, creemos que las experiencias de uso descritas aquí pueden ayudar a entender los beneficios y motivar a los constructores de herramientas NoSQL a aprovechar los principios, métodos, técnicas y herramientas de MDE. Nuestro trabajo ha sido uno de los primeros en reconocer la aparición del área de investigación de ingeniería de datos NoSQL y en contemplar la aplicación de MDE en esa área. El estudio del estado del arte realizado en esta tesis es otra contribución de esta tesis. Hemos identificado un conjunto de criterios para comparar los diferentes enfoques de inferencia de esquemas. En nuestro conocimiento, no se ha publicado ninguna revisión tan exhaustiva como la presentada aquí. Finalmente, todo el software implementado se puede descargar de la página: `https://github.com/catedrasaes-umu/NoSQLDataEngineering`.

Agradecimientos

Quiero iniciar dando las gracias al Dr. Jesús J. García Molina y al Dr. Diego Sevilla Ruiz, por haberme aceptado como tutorando, por soportarme casi cuatro años dirigiendo esta tesis que ha supuesto un maravilloso inicio hacia la investigación. He compartido muchos momentos con Jesús y le estoy muy agradecido por la enorme paciencia que siempre tuvo conmigo, ya que no sólo es un eminente profesional, sino que es una magnífica persona, muy sencilla y sobre todo muy humilde, y porque sin su magistral apoyo, simplemente esta tesis nunca se hubiera culminado. Gracias por haberme brindado su amistad, cómo olvidar tantos y tantos momentos de trabajo y de convivencia, tantas sabias palabras que siempre recibí de su parte que me sirvieron siempre de aliento en este pedregoso camino. También deseo expresar mi agradecimiento a Diego porque como director de tesis siempre estuvo pendiente de ayudarme sin escatimar esfuerzos y por haberme mostrado el camino de lo que es la buena programación, y porque al igual que Jesús me ha brindado su amistad. Gracias por haberme permitido convivir con ustedes, ya los considero parte de mi familia.

Quiero agradecer a mis padres Reyna y Juan, porque gracias a ellos, que con su ejemplo y su motivación, he logrado conseguir otra meta más en mi vida.

Quiero agradecer profundamente a mi esposa Carmen, porque siempre se ha echado la familia en hombros para sacarla adelante cuando yo no he estado con ellos. Y no sólo darle las gracias, sino también pedirle disculpas a ella y a mis hijos Reyna y Juan, por haberlos abandonado por muchos meses, quiero decirles que este logro también es suyo.

Agradezco el apoyo recibido de Alberto Hernández y Víctor M. Ruiz en algunas cuestiones de implementación.

Quiero reiterar mis más sinceros agradecimientos a Mercedes Galán y a Javier Campillo Frutos, por apoyarme siempre en las gestiones, ya que nunca me negaron su ayuda siempre que la he necesitado.

Doy las gracias a José Ramón Hoyos, a Javier Bermúdez, a Ginés García, a Rafa Valencia y a Begoña Moros, por su apoyo siempre que se los he requerido.

Agradezco infinitamente a mis amigos de la Universidad Autónoma de Guerrero y de la Facultad de Ingeniería por su apoyo incondicional: a mi compadre, el Dr. Javier Saldaña Almazán, a Margarito Radilla Romero, a Polo Bahena, a mi compadre Juan Carlos Medina, a Edgardo Solís Carmona, a Tania I. Ayala, a Joss Elizabeth, a José Luis H. H. y a Valentín Álvarez H.

No puedo dejar de agradecer a mi hermano Angelino, a mi cuñada Maga, a mi hermanas Salu, Cris, Ade, Ur, Tere y Teo, porque siempre han estado ahí cuando los he necesitado.

Cómo olvidarme de mis amigos que me hicieron sentir como si fuera parte del grupo: al Dr. Antonio García, a Antonio Gómez, al Dr. Isidro Verdú y al Dr. Domingo Giménez, gracias por su compañía y hacerme sentir menos difícil la estadía.

Inferring NoSQL Data Schemas with Model-Driven Engineering Techniques

Abstract

Modern applications that have to deal with huge collections of data have evidenced the limitations of relational database management systems. This has motivated the development of a continuously growing number of non-relational systems, with the purpose of tackling the requirements of such applications. Specially, the ability to represent complex data and achieving scalability to manage both large data sets and the increase in data traffic. The NoSQL (*Not SQL/Not only SQL*) term is used to denote this new generation of database systems.

The lack of an explicit data schema (*schemaless*) is probably the most attractive NoSQL feature for database developers. While relational systems require the definition of the database schema in order to determine the data organization, in NoSQL databases data is stored without the need of having previously defined a schema. Being schemaless, a larger flexibility is provided: the database can store data with different structure for the same entity type (non-uniform data), and data evolution is favoured due to the lack of restrictions imposed on the data structure. However, removing the need of declaring explicit schemas does not have to be confused with the absence of a schema, since a schema is implicit into data and database applications. The developers must always keep in mind the schema when they write code that accesses the database. For instance, they have to honor the names and types of the fields when writing insert or query operations. This is an error-prone task, more so when the existence of several versions of each entity is probable. Therefore, the idea is emerging of combining a schemaless approach with mechanisms (e.g. data validations against schemas) that guarantee a correct access to data. On the other hand, some NoSQL database tools and utilities need to know the schema to offer functionality such as performing SQL-like queries or automatically migrating data. A growing interest in managing explicit NoSQL schemas is therefore arising.

This thesis presents a reverse engineering strategy to infer the implicit schema in NoSQL databases, which takes into account the different versions of the entities. We call these schemas *Versioned Schemas*. We propose different schemas that can be appropriate for NoSQL databases. The usefulness of the inferred versioned schemas is illustrated through two possible applications: schema visualization, and automated code generation. In particular, we have developed database utilities to generate different soft-

ware artifacts: code for Object-Document mappers (ODM), data validators, and code to classify objects in entity versions.

The approach has been designed to be applied to NoSQL systems whose data model is aggregate-oriented, which is the data model of the three most widely used types of NoSQL stores: *document*, *key-value*, and *column family* stores. Model-Driven Engineering (MDE) techniques, such as metamodeling and model transformations, have been used to implement both the schema inference strategy and the applications, in order to take advantage of the abstraction and automation capabilities that they provide.

Thus, we show how MDE techniques can be helpful to develop solutions in the emerging "NoSQL Data Engineering" area. The schema inference approach proposed has been validated for the Stackoverflow dataset that have been stored into MongoDB.

Contents

1 Introduction 1
1.1 Motivation 4
1.2 Problem Statement 7
1.3 Research Methodology 8
1.4 Outline 11

2 Background 13
2.1 Basic Concepts of Data Modeling 14
2.2 Semi-Structured Data 17
2.3 The JavaScript Object Notation (JSON) Format 20
2.4 NoSQL databases 22
2.5 A NoSQL Database for the Running Example 25
2.6 Aggregate-Oriented Data Models 26
2.7 Object-Document Mappers 30
2.8 MapReduce Operation 31
2.9 Basis of Model-Driven Engineering 34
2.10 Tooling, Frameworks, and Languages Used 41

3 State of the Art 45
3.1 NoSQL Schema Inference 46
3.2 NoSQL Schema Representations 61
3.3 NoSQL Database Utilities 62
3.4 Final Discussion 66

4 Schemas for NoSQL Databases 71
4.1 Entities, Versions of entities and Versioned Schemas 71
4.2 A Metamodel for NoSQL Schemas 81
4.3 Overview of the Architecture 81

5 The Inference Process of NoSQL_Schema Models 85
5.1 Obtaining the Version Archetype Collection 85
5.2 Obtaining the Schema 87
5.3 Inference Validation 92

6 Visualization of NoSQL Schemas 101
6.1 Visualization of NoSQL Schemas 101
6.2 Using EMF Metamodel Diagram to Visualize Entity Union Schemas 102
6.3 Using PlantUML to visualize NoSQL Schemas 106
6.4 Using Sirius to Visualize NoSQL Schemas 113

7 Code Generation from Schema Models 121
7.1 Generating code for ODM Mappers 121
7.2 Generating other Mongoose Artefacts 127
7.3 Generation of Data Validators 129
7.4 Entity Version Classification 133

8 Conclusions and Future Work 139
8.1 Discussion 140
8.2 Contributions 145
8.3 Future Work 147
8.4 Publications 149
8.5 Software Utilities Developed 150

References 151

List of Figures

1.1 Activities in Research Methodology. 8

2.1 Example of Embedded and Referenced Objects. 16
2.2 Student Example. 18
2.3 Graph representation for the Student data. 19
2.4 JSON Grammar. 21
2.5 JSON versus XML Data. 22
2.6 Example JSON Schema. 23
2.7 Object Tree and Type Tree for a Movie. 27
2.8 Movie Database for the Running Example. 28
2.9 Object Tree and Type Tree for a Movie. 30
2.10 Schema of a Blog Post in Mongoose. 32
2.11 MapReduce word count example. 33
2.12 Four-Level Architecture for UML and ER Notations. 36
2.13 Main Elements of the Ecore Meta-Metamodel. 37
2.14 An Example Metamodel for Relational Schemas. 38

3.1 JSON Schema of our database example. 47
3.2 Four steps of the approach of Klettke et al. (extracted from [72]). 48
3.3 Structure Identification Graph for a Movie. 49
3.4 Schema management framework of Wang et al. (extracted from [119]). . . . 50
3.5 Documents stored in *Article* collection (extracted from [119]). 51
3.6 Canonical form generated for documents in Figure 3.5 (extracted from [119]). 53
3.7 eSiBu-Tree generated for documents in Figure 3.5 (extracted from [119]). . . 54
3.8 Partial Structure of a Entity Schema in MongoDB-Schema (extracted from [108]). 55
3.9 ExSchema Metalayer (extracted from [23]). 58
3.10 Generated diagram by the ExSchema tool for a MongoDB collection (extracted from [22]). 59
3.11 The SOS metalayer (extracted from [11]). 61
3.12 ER/Studio: Diagram for *Movie* collection. 63
3.13 DBSchema Result for the Running Example Database. 65

4.1 Tree for a Movie. 74
4.2 Type Tree for a Movie. 75
4.3 Schema for *Movie_4*. 76

4.4 Version Tree for *Movie_4*. 77
4.5 Entity Union Schema for the *Movie* Entity. 79
4.6 Tree of an entity union schema for Movie. 80
4.7 *NoSQL-Schema* Metamodel Representing NoSQL Schemas. 82
4.8 Overview of the Proposed MDE Architecture. 83

5.1 Schema Tree for Movie object with *_id* = 4. 88
5.2 Textual report of all the entity versions. 90
5.3 Graphical Representation of all the Entities with the sum of all fields. . . . 92
5.4 Global View of the Movie Versioned Schema (Original Model). 93
5.5 Global View of the Movie Versioned Schema (Inferred Model). 94
5.6 Graph of Times of Table 5.1. 96
5.7 Ratio of MapReduce and inference time versus number of objects from Table 5.1. 97
5.8 NoSQLSchema model result for Stackoverflow. 98
5.9 Global Set of Entity Versions Generated for Stackoverflow. 98
5.10 Union Schema using PlantUML for the Stackoverflow Example. 99

6.1 An excerpt of the diagram for Movies database generated with EMF to Graphviz. 103
6.2 MDE solution to generate schema reports and diagrams 104
6.3 Entity Union Schema visualized for the running example database. 105
6.4 Textual report of entity versions in the running example database. 107
6.5 Visualization of schema diagrams by using PlantUML. 108
6.6 Root Entity Version Schema for *Movie_4* generated with PlantUML. . . . 110
6.7 Root Entity Version Schema for *Director_2* generated with PlantUML. . . 111
6.8 Entity union schema for Movie generated with PlantUML. 113
6.9 Entity union schema for Movie generated with PlantUML. 114
6.10 Visualization of schema diagrams by using Sirius. 115
6.11 Global Schema Tree for the *Movie* example. 116
6.12 Database schema diagram for *Movies* database. 117
6.13 Plain version schema diagram for *Movies* database. 118
6.14 Nested version schema diagram for *Movies* database. 119
6.15 Entity Schema Diagram for *Criticism* 119

7.1 Overview of the Proposed MDE Solution. 122
7.2 Entity Differentiation Metamodel. 124
7.3 Excerpt of the EntityDifferentiation Model for the Example. 125
7.4 Generated Mongoose Schema. 126
7.5 Specification Example with the ODM Parameter Language. 128
7.6 Validation Code for *Movietheater_2*. 132
7.7 Validation Code for *Movietheater_1*. 133

7.8 Excerpt of the Validation Code for *Movie_3*. 134
7.9 Decision Tree Metamodel. 135
7.10 Decision Tree for the *Movie* Entity. 136
7.11 Decision Tree Code for the *Movie* Entity. 137
7.12 Check Functions for *Movie* versions 2, 3, and 5. 138

For a true writer, each book should be a new beginning where he tries again for something that is beyond attainment. He should always try for something that has never been done or that others have tried and failed. Then sometimes, with great luck, he will succeed.

Ernest Hemingway

1 Introduction

Data-intensive software applications include two main components: a set of software programs and a database. Most of these applications have traditionally used relational stores as the database. Relational database management systems (DBMS) were adopted by companies around the world more than thirty years ago.

At the beginning of the nineties, new database applications that required managing complex objects (e.g. geographic information or multimedia systems) evidenced the limitations of the relational model for the representation and processing of that sort of data. Then, new kinds of DBMS were defined, such as object-oriented and object-relational database systems. Later, the emergence of the Web also originated new challenges to the database community, which were mainly related to the management of unstructured or semi-structured data. XML-based data was proposed to address the new requirements. However these new database paradigms (object-oriented, object-relational, and semi-structured database) only achieved success in very reduced market niches, and relational systems continued being the predominant databases.

Modern applications (e.g., social media, Internet of Things, mobile apps, and Big Data) that have to deal with huge collections of data have again evidenced the limitations of relational database management systems. This has motivated the development of a contin-

uously growing number of non-relational systems, with the purpose of tackling the requirements of such applications. Especially, the ability to represent complex data, to achieve scalability to manage both large data sets, and to cope with the increase in data traffic. The NoSQL (*Not SQL/Not only SQL*) term is used to denote this new generation of database systems.

Interest in NoSQL systems databases has steadily grown over the last decade. A large number of companies have already embraced NoSQL databases, and the adoption will rise considerably in next years, as reported in [1, 87]. The "nosql-database.org" website shows a list of about 225 existing NoSQL systems. Actually, the NoSQL term refers to a varied set of data modeling paradigms that manage semi-structured and unstructured data. The major NoSQL categories are: *document*, *wide column* and *key-value* stores, and *graph-based* databases. Except for graph databases, the paradigms aim to represent semi-structured data using an aggregate-based data model [109]. These paradigms are the most widespread, being MongoDB [80] the most widely used NoSQL system [88]. The MongoDB company was recognized as a *Leader* by the Gartner 2015 Magic Quadrant for Operational Database Management Systems, and other companies that offer aggregate-based systems, such as Redis [97] or Couchbase, were considered *Leader* and *Visionary*, respectively, in that report.

It is worth remarking that systems in the same NoSQL paradigm can even have different database features. However, most NoSQL systems have a few common properties, namely: SQL language is not used, schemas have not to be defined to specify the data structure, the execution on clusters is the main factor that determines its design, and they are developed as open-source initiatives [109]. The ability of storing data without prior definition of a schema is one of the most attractive features of NoSQL systems although originates some drawbacks as discussed later in this chapter.

The recent Dataversity report "Insights into Modeling NoSQL" [1] has remarked that data modeling will be a crucial activity for NoSQL databases and has also drawn attention on the need for NoSQL tools that provide functionality similar to those available for relational databases. The authors of this report identified three main capabilities to be offered by NoSQL modeling tools: model visualization, code generation, and metadata management. These three kinds of capabilities requires the knowledge of a schema as stated in that report.

Data Engineering is the Computer Science discipline concerned with the principles, techniques, methods, and tools to support the data management in the software development. Data are normally stored in database management systems (e.g. Relational, Object-oriented, or NoSQL) and Data Engineering has been mainly focused on relational data so far, although interest is shifting towards NoSQL databases. In this thesis, we have addressed issues which are related to topics of Data Engineering, more specifically to NoSQL data reverse engineering and the development of NoSQL database utilities able of providing capabilities of the three categories highlighted in the report mentioned above. The work of this thesis is therefore focused on *NoSQL Data Engineering* that is an emerging research area within of the *Data Engineering* field.

Over the last few years, *Model-driven Software Engineering* (MDSE or simply MDE) is increasingly gaining acceptance, mainly owing to its ability to tackle software complexity and improve software productivity [110, 121, 16]. MDE promotes the systematic use of models in order to raise the level of abstraction at which software is specified, and to increase the level of automation in the development of software. MDE techniques, more specifically metamodeling and model transformations, have proven to be useful for forward and reverse engineering tasks.

Data schemas are models, and operations on them can be implemented using model transformations. Transformational approaches have therefore traditionally been used to automate data engineering tasks such as normalisation, schema conversion, or schema integration as explained in detail in [56]. The implementation of such tasks could be facilitated by MDE. The rationales should be sought in MDE, which provides a specific technology (principles, techniques and tools) with which to build transformational solutions. However, data engineering community has paid little attention to the application of MDE as noted in [105].

The purpose of this thesis is to define a model-driven reverse engineering strategy to infer implicit schemas from NoSQL databases, and to explore the usefulness of the inferred schemas by developing some database utilities. Model-driven solutions have been devised to tackle the implementation of utilities for three applications: graphical schema representation, automated generation of data validators and code for ODM mappers, and database object classification.

The rest of this chapter is organized as follows. First, the work is motivated in Section 1.1. Then, the problem and goals are stated in Section 1.2. Afterwards, Section 1.3 describes the research methodology applied in this thesis. Finally, the structure of this document is outlined.

1.1 Motivation

While relational systems require the definition of the database schema in order to determine the data organization, in most NoSQL databases data is stored without the need of having a previously defined schema. This lack of an explicit data schema (*schemaless*) is probably the most attractive NoSQL feature for database developers. Being schemaless, a larger flexibility to manage data is provided, for instance the database can store data with different structure for the same entity type (non-uniform data), and data evolution is favored due to the lack of restrictions imposed on the data structure. However, removing the need of declaring explicit schemas does not have to be confused with the absence of a schema, since a schema is implicit into stored data and database application code. The developers must always keep in mind the schema when they write or maintain code that accesses the database. For instance, they have to honor the names and types of the fields when writing insert or query operations. This is an error-prone task, more so when the existence of several versions of each entity is probable. On the other hand, some NoSQL database tools and utilities need to know the schema to offer functionality such as performing SQL-like queries or automatically migrating data or either manage data efficiently. A growing interest in managing explicit NoSQL schemas is therefore arising, as evidenced in the schema inference approaches recently proposed [72, 102, 119] and a few tools already available to help to NoSQL developers [81, 108, 47]. Regarding to the difficult of writing code that correctly accesses to data, two strategies are emerging for NoSQL database applications: (i) combining the schemaless approach with mechanisms that guarantee a correct access to data (e.g. data validators) [52], and (ii) using mappers which converts NoSQL data into objects of a programming language. Most current mappers are for document stores (Object-document mappers, ODMs), and Mongoose [81] is the most widely used ODM, created for MongoDB and Javascript. However, ODMs for other languages, such as Java and PHP, are also available.

As noted above, the schemaless feature allows non-uniform data for entities, i.e. an entity

can have different versions of the implicit schema (type) that characterizes the data stored of that entity. Each version is defined by a set of attributes. The version of an entity can be originated, for instance, by a initial design choice or changes on the implicit schema of an entity (objects that have the evolved schema coexist with objects that have the new schema). Therefore, taking into account that each entity can have one or more versions, establishing the schema (or type) of an entity requires to consider the set of schemas of its versions. Usually, the union of all entity versions [72] or some form of approximate type [119] is considered the schema of an entity. A notion of global database schema for NoSQL databases has not been proposed yet as far as we know.

The aforementioned Dataversity report discussed the implications of the absence of explicit schemas in NoSQL systems and considers that "Data isn't an asset until meaning can be extracted from it", and without a clear understanding of the inherent structure in NoSQL database, "all the Big Data in the world is useless", and "There are many emerging techniques and technologies that allow for the modeling of NoSQL data stores, especially with reverse mapping techniques, but they are still maturing". It is worth noting that this report was published in mid-2015, while our work started at the beginning of 2014.

The authors of the report surveyed to industry's experts whose job function is related to the data management: Data and/or Information Architecture (49%), Executive Management (8%), IT Management and Software/System Vendor (6,8%), and Business Intelligence and/or Analytics (6,1%). The survey consisted of 20 questions about the NoSQL implementation in the enterprise, NoSQL models, tools and functionality. This survey highlighted that the adoption of NOSQL is still limited (25% of respondents indicated that their enterprises was using NoSQL databases) but the expected growing rate is high (more than 60% of respondents indicated that they planned to implement document or graph stores in less than 18 months). With regard to the use of models, NoSQL modeling is considered a needed practice (63% indicated that data modelers are needed), although no modeling is currently done in many companies (a third of respondents said that they only write application code).

From this survey, the authors analyzed the role of modeling for NoSQL stores. Next, we will comment some of the more relevant results or its analysis in relation to the aims of our work. The report highlighted that data model tools are essential to tackle some of

the main challenges for the industrial adoption of NoSQL systems as: data modeling, data governance, documentation and appropriate tooling. There is a lack of NoSQL database tools providing capabilities existing for relational databases. Moreover, existing tools are immature. The report identified three main categories of desired functionalities for NoSQL systems: *diagramming*, *code generation*, and *metadata management*. Model visualization, documentation and reverse engineering of existing NoSQL database are noted as features to be provided by diagramming tools. Visual diagrams would be useful in designing databases, making decision, documenting, and representing the schemas inferred by means of reverse engineering processes. The report noted that NoSQL databases are schemaless, and therefore a reverse engineering process is needed to discover the implicit schema. Regarding to code generation, models could be used to automatically generate artefacts of applications, i.e. application code could be generated from high-level NoSQL schemas or physical schemas could be generated from conceptual or logical schemas. This code generation would be useful to increase productivity and improve quality (e.g. number of errors is reduced), among other benefits. Finally, metadata are considered essential in some scenarios as achieving the integration among different NoSQL stores. It is worth note that information expressed in schemas is the essential part of metadata managed in databases. The authors of the report remarked that persistence polyglot will be the norm in a very near future. This will demand the integration of database systems of different technologies. Models could be useful for such an integration, and database tools will have to be able to manage models for different paradigms.

When the goals of this thesis were established at the beginning of 2014, no works on schema inference for NoSQL systems had been published yet, and no NoSQL database tools were available. Therefore, our work has been carried out without knowledge of other previous related research efforts. For example, an article describing our first version of the inference process was submitted to CIbSE'2105 in December, 2014 [82], and the second version was described in an articles that we submitted to ER'2015 in April, 2015 [102]. These works already presented some solutions for schema visualization and generation of code for ODM mappers. The first schema inference approaches proposed by other research groups were also published throughout the year 2015 [72, 119, 23], as well as the first tools of NoSQL schema visualization [47] and ODM mappers [81].

1.2 Problem Statement

The absence of explicit schema is a feature needed in NoSQL systems because it provides the required flexibility when the data structure varies often. However, developers need to understand the implicit schema when they write code, and some tools require knowing the schema to support some functionality. This demands the definition of reverse engineering strategies aimed to discover the inherent schemas, and building tools that take advantage of the inferred schema to offer capabilities that assist developers. These tools will bring the benefits of schemas without losing the flexibility gained by being schemaless. The existence of entity versions in NoSQL databases is the main challenge to be addressed in the schema inference process. With regard to the implementation of the inference process and the database tools, we believe that MDE techniques, especially metamodeling and model transformations, ease this task. Metamodeling is useful to build representations at high level of abstraction of the information being managed, and model transformations help to automate the development. The work of this thesis will be focused on NoSQL systems whose data models are aggregate-oriented.

The goals of this thesis are the following:

Goal 1. Design and implementation of a schema inference process. To devise a model-driven reverse engineering approach to infer the implicit schemas in NoSQL databases. The strategy defined should take into account the existence of entity versions and obtain a model that represents all the versions existing in the database for each entity.

Goal 2. Proposal of a notion of NoSQL data schema. To conduct research on how the traditional concept of relational database conceptual schema can be defined for NoSQL databases. The definition should include entities (or entity versions) and relationships between them, in particular aggregation and references.

Goal 3. Design diagrams for NoSQL schemas and implement tools that support its visualization. To investigate what kind of diagrams could visually represent the different kinds of schemas identified for NoSQL databases. We should also implement some diagramming tools that support the visualization of these diagrams. MDE techniques would be used to implement these tools.

Goal 4. Code generation for ODM mappers. To develop an MDE solution to automatically generate code for ODM mappers, such as schemas and validator functions for Mongoose.

Goal 5. Generation of data validators. To develop an MDE solution to automatically generate validators (i.e. schema predicates) intended to be used to check that data are stored without violating the schema.

Goal 6. Definition of a strategy to classify objects. To define a clustering algorithm to determine which entity version an object belongs to. This classification could be useful, for instance, for building migration procedures to a higher level.

The last four goals has allowed us show the usefulness of the inferred schemas.

1.3 Research Methodology

In order to achieve the objectives of this thesis that were introduced in the section above, we have followed the design science research methodology (DSRM) described in [68, 117]. The design process consists of the six activities shown in Figure 1.1: (1) Problem identification and motivation, (2) Define the objectives of a solution, (3) Design and development, (4) Demonstration, (5) Evaluation, and (6) Conclusions and communication.

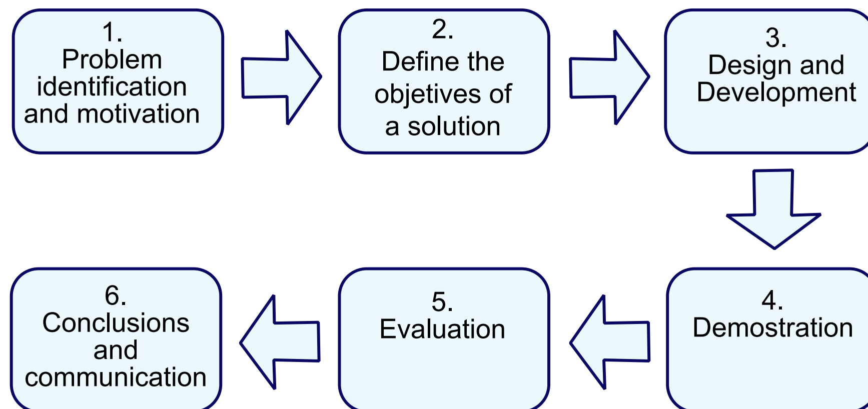

Figure 1.1: Activities in Research Methodology.

This is an iterative process in which the knowledge produced throughout the process by constructing and evaluating new artifacts serves as feedback for a better design and imple-

mentation of the final solution. Following the activities defined in DSRM, we began by identifying the problem and its motivation. The ideas that served to establish the purpose of this thesis came from our experience in model-driven data and code reverse engineering [124, 105], and our vision of the need or convenience of schemas for developing some applications involving NoSQL systems, such as the migration of legacy relational systems to NoSQL systems. The Dataversity report [1] and the first research efforts [72, 119] as well as the first tools of NoSQL schema visualization [47] and ODM mappers [81] corroborated our hypothesis.

In previous Section 1.1, we have motivated the problem and explained how it was identified, and we have clearly stated the goals in the previous Section. They basically are: (i) to define a reverse engineering strategy to infer the implicit schema in NoSQL databases, which takes into account the different versions of the entities, and (2) to illustrate the usefulness of the inferred schemas through three possible applications: schema visualization, and automated generation of data validators, and code for ODM mappers. Normally, a knowledge of the state of the art is required to state the problem and the consequences (i.e. benefits and drawbacks) of its solution. In our case, the state of the art has been studied as we progressed in our work. For example, we knew of the approaches [72] and [119], once we had designed and implemented our inference schema strategy. As can be observed in Chapter 3, most of discussed research works that have to do with NoSQL systems were presented after the beginning of our thesis work.

We started the third activity by designing a schema inference strategy. Then we built a rapid prototype as proof of concept of our approach. We defined an MDE solution that, in a first step, injected all JSON stored objects into JSON models, and then a RubyTL [34] model-to model transformation generated a model that represented all the entity versions extracted. The injector was generated through EMFText [49] that is a tool for defining textual Domain-Specific Languages (DSL), i.e. the input would be the JSON grammar in the required format by the tool. This first version was presented in the CIbSE'2015 conference [82]. A crucial element of this first version was the NoSQL_Schema metamodel defined to represent the inferred versioned schemas. Then, we observed that a Map-Reduce operation could improve significantly the performance by selecting a unique stored object for each entity version. Moreover, a general-purpose language could be more convenient

than model-to-model transformation languages to implement the reverse engineering strategy because of the complexity of the task. This second version of the inference process was presented in the ER'2015 conference [102]. In that article, we illustrated the usefulness of the inferred versioned schemas through two possible applications: schema visualization, and automated generation of data validators. After, we improved the performance of the schema inference process and performed a validation by creating a MongoDB database with the Stackoverflow dataset [113].

Regarding the schema visualization, we first devised the solution proposed in [102], which is based on generating an Ecore metamodel from the extracted schema model. After, we experimented with UML class diagrams generated as PlantUML code [96], and then we noted the need of generating specific diagrams for NoSQL schemas. This specific notation was developed as part of a master's thesis [29] in July 2016, and presented in the JISBD'2017 conference. The MDE solution designed to generate data validators is based on a decision tree that reduces the number of checking needed to find the version which a data object belongs to. We are now writing a longer article that describes the finally applied schema discovering strategy and the applications of inferred schema visualization, the validation with the Stackoverflow database, and the generation of data validators. We have also designed an MDE solution to generate code for ODM mappers (schemas and other artefacts depending on the mapper target), in particular we have considered the Mongoose [81] and Morphia [83] mappers. Our work on generating code for ODM mappers was presented in the Modelsward'2107 conference [104].

All the approaches that we have defined are based on MDE technology because this technical space provides principles, techniques and tools that (i) facilitate the involved information representation at a high level of abstraction by using models and metamodels, and (ii) automate the generation of software artifacts from models by means of model transformations.

In the fourth activity, we demonstrated that each of the solutions developed adequately works. We have created a small movie database to test our implementations, and we have validated our schema discovering approach by using a MongoDB database created from the Stackoverflow dataset [113]. The Stackoverflow has been also used to validate the scalability of the inference in big databases. It is worth remarking that we have applied an iterative

method, and the artifacts (e.g. metamodels and inference strategy) developed have evolved throughput the work of this thesis. For example, in the validation of the inference strategy, we realized that more heuristics to identify references were needed.

Finally, we have evaluated our work by comparing the objectives with the results obtained. We have concluded by identifying some improvements and future research lines.

1.4 Outline

The structure of the rest of this document is as follows:

- **Chapter 2** introduces the background needed for a better understanding of this thesis. It presents concepts related to data modeling, NOSQL systems, and the MDE paradigm.
- **Chapter 3** analyzes the state of the art in three areas. Namely, schema discovering approaches, unified representation of NoSQL data models, and schema diagramming tools. The lacks and weaknesses of each work are discussed and several dimensions are defined to compare the works.
- **Chapter 4** defines the different kind of schemas that we have considered for NoSQL. In addition, the NoSQL_Schema metamodel defined to represent the inferred schemas is described, and a overview of the schema discovering strategy is outlined.
- **Chapter 5** explains in detail the reverse engineering process defined in this thesis to discover implicit eschemas in NoSQL databases. It also includes the validation of the proposal by inferring the schema for the Stackoverflow dataset.
- **Chapter 6** describes the different kinds of diagrams used to visualize the inferred schemas: UML class diagrams for Ecore metamodels, UML class diagrams generated with PlantUML, and diagrams of a notation specifically defined for this purpose. The implementation of each of utilities developed is explained.
- **Chapter 7** describes the MDE solutions designed and implemented to generate data validators and code for ODM mappers. The strategy devised to classify objects into versions of entities is also explained.

- **Chapter 8** concludes this thesis by analyzing to what extent the goals we have presented in this chapter have been achieved. We summarize the main contributions and publications of this thesis. We end by providing some insights into further work.

If I have seen further than others, it is by standing upon the shoulders of giants.

Isaac Newton

2

Background

This chapter introduces the background needed for the better understanding of this thesis. The explanations given concern to the aims of our work and the technologies used in the implementation of the developed tools. The former will also help to motivate our work. Firstly, we will introduce some basic concepts in data modeling; we will distinguish between *reference* and *aggregation* relationships. Next, the notion of *semi-structured data* is defined, and the JSON format is presented. Then, we will describe the main *NoSQL paradigms*. At this point, we will present a JSON database that will be used as a running example throughput this document. After, we will explain a notion that is essential in our work: *aggregate-oriented data model*. *Model-Driven Software Engineering* (*MDE*) techniques have been used to implement the tooling developed in this thesis. We will therefore introduce the central elements of MDE: metamodels, model transformations, and domain-specific languages. Finally, as we have to deal with databases that may have potentially millions of objects, we have to develop our algorithms in a scalable way. We used MapReduce techniques [39], implemented in most NoSQL databases, that are scalable by design.

2.1 Basic Concepts of Data Modeling

In software development, models are useful because they (i) help reasoning about the system to be implemented, (i) serve as documentation of the system being built and the design choices made, (iii) facilitate the communication among stakeholders, and (iv) allow automatically generating the code of the final system. For example, *UML class models* [107] are created to represent the structure of the domain classes of an object-oriented application. In the database area, models are also created for the same purposes. For instance, *Entity-Relationships* (E/R) models [26] are used to create database schemas.

The *data model* term denotes the formalisms (i.e. languages) whose purpose is to represent the structure of the data managed by an information system [111]. The *schema* term is used to refer to the representations (i.e. models) expressed by means of a data model. Data models (and therefore schemas) can be expressed at three levels of abstraction, according to the ANSI/SPARC architecture [25]: conceptual, logical, and physical. A *conceptual* schema expresses the semantics of a domain: entities and relationships between them. A *logical* schema is expressed in terms of a particular database paradigm (e.g. relational, object-oriented, or graph.) A logical schema can be derived from a conceptual schema by transforming entities and relationships into elements of the target paradigm (e.g. tables in relational databases, and classes in object oriented databases.) Data stored in a database conforms to a logical schema which determines its structure. The *database model* term is frequently used to refer to logical data models. Finally, a *physical* model expresses details of how data are physically stored (e.g., indexes and clusters.) For instance, in relational database design, the E/R and UML modeling languages are widely used to create conceptual, logical, and physical schemas. In this thesis, we are interested in schemas and models for NoSQL databases. We will use UML-like representations to express the NoSQL data schemas.

Throughput this document, we shall use the *database schema* term to refer to logical schemas created using a data modeling language, and the *data entity* (or simply *entity*) term with its usual meaning, i.e. category of physical objects or concepts of the real world that are represented in database schemas. So, a database stores information of entities in some format. For instance, a relational database of an university can store information on doctoral student entities (e.g. name, email, and supervisors) in form of rows and tables. Entities have attributes or properties whose values are of a simple type (e.g. integer, float, and string), and

there are relationships between entities (e.g. a student *has* one or more supervisors, and a student name *is formed by* a first name and a last name).

In conceptual modeling, a relationship between two entities is usually represented by means of an *association* relationship. Moreover, an *aggregation* relationship is used to model *part-of* relationships. For instance, UML provides associations that can be labeled as *composite* in order to represent an aggregation relationship in which the parts are dependent and exclusive to the whole. These two relationships are implemented by means of *object references* in most object-oriented databases and programming languages. References are manipulated in a transparent way to developers, which can navigate among objects by using high-level constructs.

The *embedded object* technique can also be used to implement a relationship between two objects, specially if it is an aggregation. For instance, Figure 2.1 shows a *Movie* object of the database that will be introduced as running example in Section 2.5. This figure shows how the values of the *rating* and *criticisms* properties are embedded objects. In particular, the *Movie* object has embedded a *Rating* object and an array of *Criticism* objects. It is worth noting that these embedded objects are not implementing an aggregation (i.e. part-of) relationship defined at the conceptual level.

This technique is normally used to express that the objects involved into a relationship are stored as a single unit, instead of being referenced. In fact, the embedded object technique usually implies that developers must explicitly manage references. That is, they must choose if a relationship is implemented by means of either references or embedded objects. In our example, the value of the *director_id* property of *Movie* is a reference to a *Director* object. This decision would have been taken by considering criteria as efficient access to the information of embedded objects. Therefore, the embedded object relationship conveys semantics that can be used with different purposes, such as allocating memory or disk space to objects, or assuring consistency when objects are changed (e.g. a cascading delete if the root object is removed).

The *aggregate* term is normally used to refer to the object structure that consists of a root object that recursively embeds other objects, so that an aggregation hierarchy is formed (e.g. *Movie* object in Figure 2.1.)

While complex data is represented in relational databases through joins by means of

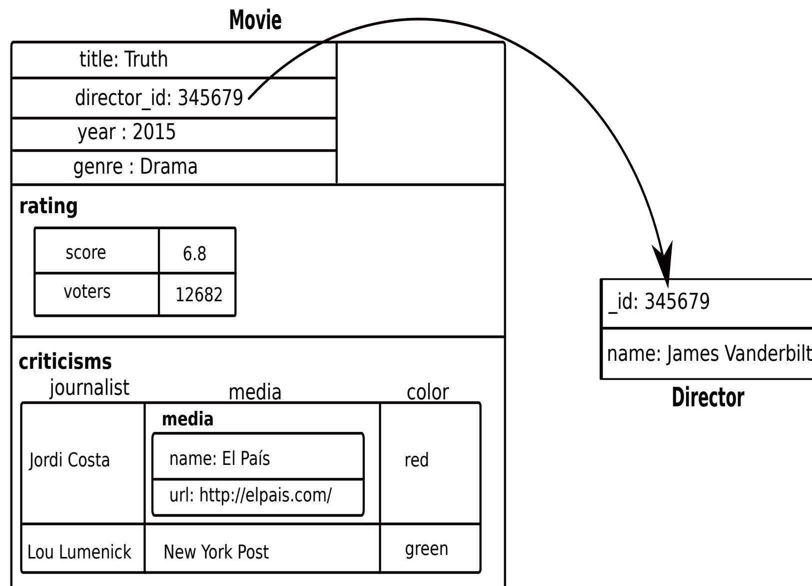

Figure 2.1: Example of Embedded and Referenced Objects.

foreign keys (i.e. references between tables), object references (whose representation is not managed by developers) and aggregate objects provide more abstract forms of representing such data. Reference and aggregation constructs facilitate the representation and manipulation of relationships among objects.

The distinction between object references and embedded objects has been supported by different object-oriented databases and languages, but it has not become part of the mainstream software development. The Eiffel language [77] distinguished between references and expanded types in order to allow the definition of object references and embedded objects, respectively. The aggregation relationship was supported by some object-oriented databases [13, 24], with the purpose of representing composite objects (i.e. the *part-of* relationship). As explained in Section 2.9, some object-oriented metamodeling languages, such as Ecore [114] and MOF [92], have also distinguished between references and composite references (i.e. embedded objects) in order to establish the relations among concepts of the abstract syntax of a language. In the software design realm, the use of aggregate objects is proposed in the Domain-Driven Design method [50] in order to assure the application consistency by accessing aggregated objects only through the root object that controls the changes so that they are correctly applied. Note that commonly used programming lan-

guages (e.g. Java, Ruby, and C#) do not offer support for aggregates, hence the developers have the responsibility of writing the code aimed to ensure that a set of objects behaves as an aggregate.

Although the distinction between object references and embedded objects is not supported in most of languages and systems, more widespread NoSQL database systems are based on an aggregate-oriented data model [109], as explained in detail later in this chapter. Therefore, such a distinction will play an essential role in NoSQL data models that we will address in this thesis.

2.2 Semi-Structured Data

In the mid 1990s, the notion of *semi-structured data* arose to define the properties of data involved in the Web and the new applications that appeared around it (e.g. genome databases or geographical databases), as well as to tackle the data integration of independent sources and data browsing (i.e. to write data queries without knowledge of the schema) [8, 20, 9].

Semi-structured data is mainly characterized by the fact that its structure is not defined in a separated schema, but it is implicit on the data itself. A semi-structured data is usually described as a tuple of *key-value pairs*. Keys (a.k.a. fields and tags) denote *properties* or *attributes* of the data, and the values can be primitive (i.e. atomic values of types such as numbers, string, and boolean) or complex (tuples and arrays). Figure 2.2 shows how data of a doctoral student could be expressed by using the syntax defined in [9].

A piece of data on doctoral students includes two fields with primitive values (*phone* and *email*) and three fields with complex values (*name*, *supervisors*, and *program*). The *name* field is a tuple with the first and last names. The *program* field is a tuple that registers the title of the doctorate study program and the faculty in charge of this program. The *supervisors* field registers the array with the identifiers of the tutors of the doctoral student. Each identifier would be a *reference* to the corresponding piece of data with information about a supervisor. Semi-structured data has, therefore, a hierarchical structure (i.e. nested structure) with a root tuple (student in the example) which can include other tuples and arrays. It is worth noting that semi-structured data is a format to express embedded or aggregate objects but without the necessity of a prior definition of a schema. Hereafter we will use the term *object* instead of *tuple*.

```json
{
    "student": {
        "email": "severino.feliciano@um.es",
        "name": {
            "firstName": "Severino",
            "lastName": "Feliciano"
        },
        "phone": 123456789,
        "program": {
            "centre": "Faculty of Computer Science",
            "title": "PhD in Computer Science"
        },
        "supervisors": [
            99988877,
            22233344
        ]
    }
}
```

Figure 2.2: Student Example.

As indicated in [20], a piece of semi-structured data can be formalized as some kind of graph-like or tree-like structure. If references among pieces of data are allowed, the structure would be a graph. Figure 2.3 shows the graph for the student data example. We suppose that supervisors have the *name* and *department* fields among others. Our graph representation includes leaf nodes labeled with meaningful data (atomic values) and intermediate edges labeled with symbols that denote the field names. An array is represented by labeling internal edges with integers. A root or intermediate node has a child node by each field of the object associated. As indicated in [20], this graph model can be defined as:

```
type base = int | float | string | ...
type tree = base | set (tree x symbol)
```

The creation of runtime and persisted data usually requires the existence of a specification of its structure. For instance, objects are instantiated from classes (i.e. types) during the execution of object-oriented programs, and a schema must be defined prior to storing data in a relational database. However, semi-structured data can be directly created, because it includes information that describes its structure. This is why it is commonly characterized as "schemaless" and "self-describing". In the previous example, the information on the structure is within the data (i.e. graph shown in Figure 2.3), but no schema or types have been

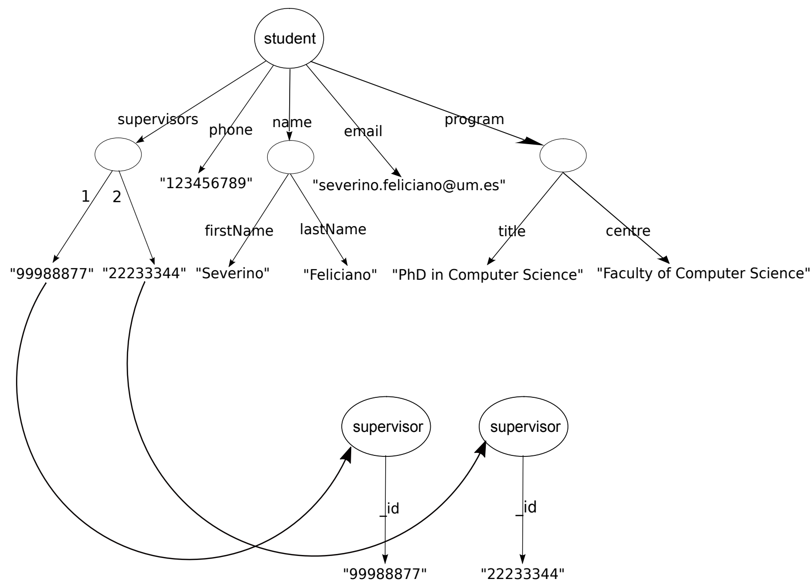

Figure 2.3: Graph representation for the Student data.

defined in a separate specification separated of the data.

The lack of explicitly defined schemas (*schemaless*) offers some significant benefits to developers of database applications as noted in [8] and [109]. In particular, this characteristic facilitates the data evolution and having non-uniform data.

Data evolution Data structure can evolve frequently as schema changes are not needed. Data with the new structure can be stored without any kind of restriction. In our example, student data that registers more than one phone number could be stored at any moment. In relational systems, data evolution requires to change the schema and some kind of data migration.

Non-uniform data One of the main strengths of semi-structured data is to allow the variation in structure on data of the same type. For instance, the student data could be registered with variations: a property could have values of different type (e.g. the name is either a single string or the first and last name are distinguished in a nested object) or either a property is optional (e.g. when a student complete his doctoral thesis at a company, some information on the involved company could be registered). These variations typically are minor changes in representation.

Although the absence of a schema provides a greater flexibility in managing data, however it has some drawbacks for developers. When a schema is formally defined (e.g. a relational schema), a static checking assures that only data that fits the schema can be manipulated in application code, and mistakes made by developers in writing code are statically spotted. In fact, the analogy to statically and dynamically typed languages is commonly used to note the difference among semi-structured data and data that conforms to a schema [20]. When a schema is not defined, errors such as duplicated or missing properties are not catched when the data are created.

As explained in Chapter 1, most NoSQL databases are schemaless because (i) storing data whose structure is rapidly changing or is unknown in advance, and (ii) supporting agile development are two of the main requirements that they must satisfy. New application domains in which data structures are expected to experience rapid change were already identified twenty years ago in the first works on semi-structured data [8, 20]. The number of such applications has grown considerably since then.

XML has been the commonly used format to store semi-structured data from the advent of Web and its adoption by the World Wide Web Consortium (W3C) as a standard to exchange data on the Web. However, since the appearance of JSON (JavaScript Object Notation)* at the end of the last decade, XML is losing predominance as data interchange format in favor of JSON. In fact, JSON is the format used to store data in majority of NoSQL databases (data are internally encoded in some binary serialization format such as BSON). Next, we shall describe the JSON format.

2.3 The JavaScript Object Notation (JSON) Format

JSON [3] is a standard human-readable text format widely used to represent semi-structured data. This notation is taking the place of XML as primary data interchange format because it is more simple and legible. JSON is a small subset of JavaScript, more specifically JSON data are JavaScript literals. This has significantly contributed to the success of JSON. There are two JSON standards: ECMA and RFC 7159. The main difference is that ECMA considers any JSON value as a valid JSON text, instead RFC 7159 establishes that a valid JSON

* http://www.json.org/.

text must have an object or array as root. JSON is highly interoperable because data interchanged are simply Unicode text [112].

Figure 2.4 shows the JSON grammar in form of syntax diagrams which are taken from [3]. This grammar specifies that a JSON text (a.k.a. document) can be either (i) an *object* formed by a set of key-value pairs or (ii) an *array* (i.e. an ordered list) of values. The type of a JSON *value* may be a primitive type (Number, String, or Boolean), an object, or an array of values. `null` is used to indicate that a key has no value. This grammar allows to represent data in form of nested structures by using objects (i.e. tuples) and arrays (i.e. lists) as composition constructs. Note, that JSON is a notation intended to express semi-structured data (tree-like form and schemaless). In fact, example of Figure 2.2 is really a valid JSON text. According to its tree-like structure, we will distinguish between root (e.g. *student* object) and nested objects (e.g. the *name* and *program* objects nested to *student*) in JSON documents.

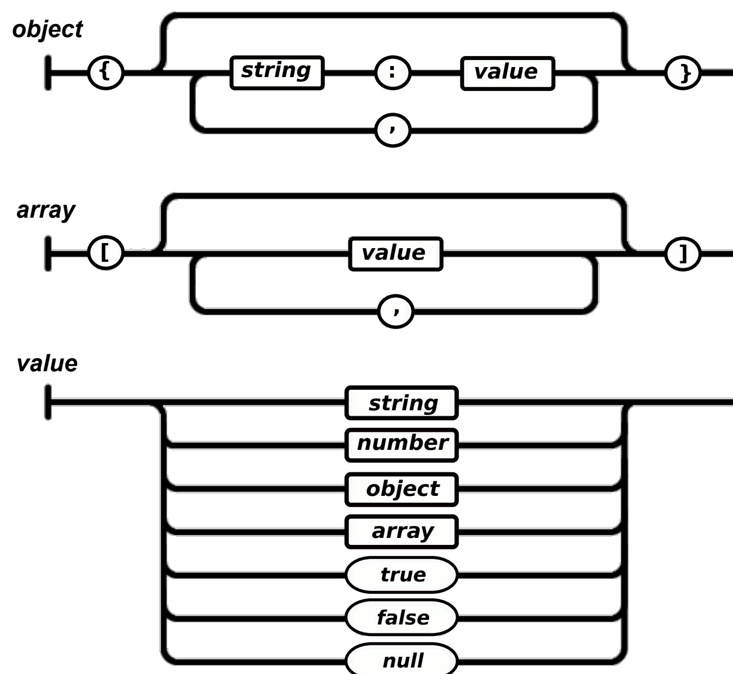

Figure 2.4: JSON Grammar.

Figure 2.5 shows how the example of doctoral student data could be expressed in JSON and XML in order to contrast both formats. This example evidences the benefits of legibility and simplicity commonly mentioned to explain that JSON is replacing to XML as main

data interchange format.

XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
  <student>
    <name>
      <firstName>Severino</firstName>
      <lastName>Feliciano</lastName>
    </name>
  <phone>123456789</phone>
  <email>severino.feliciano@.um.es</email>
  <supervisors>99988877</supervisors>
  <supervisors>22233344</supervisors>
    <program>
      <title>PhD in Computer Science</title>
       <centre>Faculty of Computer Science</centre>
    </program>
  </student>
```

JSON

```json
{
  "student":{
    "name": {
      "firstName": "Severino",
      "lastName": "Feliciano"
    },
    "phone": "123456789",
    "email": "severino.feliciano@.um.es",
    "supervisors": ["99988877", "22233344"],
    "program": {
      "title": "PhD in Computer Science",
      "centre": "Faculty of Computer Science"
    }
  }
}
```

Figure 2.5: JSON versus XML Data.

JSON Schemas The *JSON Schema* initiative [4] has recently emerged to provide standard specifications for describing JSON schemas. Although its adoption is still very limited, some tools (e.g. validators, schema generators, documentation generators) have evidenced the usefulness of having JSON schemas. Figure 2.6 shows an example of a JSON schema for the *Student* data introduced above. The JSON type of each field is specified. For an *Object* field, the fields ("properties" field) are specified, and the required fields are enumerated ("required" field). For an *Array* field, the type of the items is specified, and the properties required are enumerated.

2.4 NoSQL databases

As commented in Chapter 1, the NoSQL term is used to refer to different new database paradigms which are an alternative to the predominant relational DBMSs. Web applications such as social networks (e.g. Facebook), text searching (e.g. Google) and e-commerce (e.g. Amazon), which manage very large and complex data, are some examples of scenarios where different NoSQL systems have been successfully used. The main difference between NoSQL databases and relational databases is the set of properties they provide; while relational databases provide all the ACID (Atomicity, Consistency, Isolation and Durability)

```
{
    "type": "object",
    "properties": {
        "name": {
            "type": "object",
            "properties": {
                "firstName": "string"
                ,
                "lastName": "string"
            },
            "required": [
                "firstName",
                "lastName"
            ]
        },
        "phone": "string",
        "email": "string",
        "supervisors": {
            "type": "array",
            "items": {
                "type": "string"
            }
        },
        "program": {
            "type": "object",
            "properties": {
                "title": "string",
                "centre": "string"
            },
            "required": [
                "title",
                "centre"
            ]
        }
    },
    "required": [
        "name",
        "phone",
        "email",
        "supervisors",
        "program"
    ]
}
```

Figure 2.6: Example JSON Schema.

properties, NoSQL databases provide a subset of the CAP properties: Consistency (whenever a writer updates, all readers see the updated values), Availability (the system operates continuously even when parts of it crash) and Partition tolerance (the system copes with dynamic addition and removal of nodes) [109].

Actually, the NoSQL term refers to a varied set of data modeling paradigms aimed to manage semi-structured and unstructured data. Most of them have a few common properties, namely: SQL language is not used, schemas have not to be defined to specify the data structure, the execution on clusters is the main factor that determines its design, and they are developed as open-source initiatives. The major NoSQL categories are: *document*, *wide column* and *key-value* stores, and *graph-based databases* [109].

Key-value databases Provide the most simple way to storing data: a key-value pair. The database is therefore a set of key-values pairs. This structure is similar to the *map* collection type that is provided in many programming languages. Most of key-value systems do not assume any structure on the data, and treats it as blobs of information

(e.g. Riak [100]), but a few of them allows assign a type (e.g. integer) to values (e.g. Redis [97]).

Document databases Data are also stored as a set of key-value pairs, but the value takes form of an structured document (normally a JSON-like document) that can be navigated to obtain particular data or to form queries. In addition, the database is usually organized into a set of collections, and each collection contains the documents stored for an kind of entity (e.g. student or movie). Then queries can be issued on collections. MongoDB [80] and CouchDB [5] are the most used document systems.

Wide Column databases They are organized as a collection of rows, each of them consisting of a row key and a set of column families, each of them being, in turn, a set of key-value pairs. Cassandra [21] and Hbase [57] are the most used column-family systems.

Graph databases A database is stored as a *labeled property graph*; nodes contain entity's properties, and edges represent the relationships among entities. Edges can also be labeled with properties. Unlike other NoSQL systems, relationships are first-class citizens in graph databases. Neo4J [85] and OrientDB [94] are two well-known graph databases.

While graph databases are based on a graph data model which emphasizes the relationships among data entities, the other three categories (key-value, document, and wide column) are based on a aggregate-oriented data model which emphasize how the data entity has a nested structured which results of the fact that entities have properties whose value can also be entities (i.e. the tree-like structure proper of semi-structured data) [109].

Except for a few exceptions like Cassandra, most NoSQL systems are schemaless. As explained in Section 2.2, this is a significant difference with relational databases, and provides greater flexibility, as the structure of the data stored is not restricted by a schema.

It is really complicated to determine what NoSQL database best fits the needs of a company, given the number of existing systems (225 in "nosql-database.org") and that they considerably differ even within the same category. This has motivated the publication of guides for helping to choose the most appropriate NoSQL system for a particular set of requirements for a data-intensive application. In this thesis, we are only interested in data modeling

issues, and therefore issues on performance and CAP properties will be not addressed in this section.

2.5 A NoSQL Database for the Running Example

Here, we introduce a set of JSON documents that represent data on movies, which will be used throughput this thesis as an running example of database. For the purposes of this thesis, we have considered a NoSQL database as an arbitrarily large array (i.e. a collection) of JSON objects that include: a) a field (e.g. *type*) that describes its entity type; and b) some form of unique identifier for each object (in our case the *_id* field). This format is non-compromising, and provides system independence. In fact, it is very similar to what it is actually used in most NoSQL database implementations. For example, CouchDB guides recommend the usage of the *type* field. MongoDB creates one collection for each type of object, so that the collection name could provide the *type* field. In HBase, the *type* field of an object could be the name of its *column family*. If the value of the *type* field is not directly obtainable, some heuristics could be used. However, in some cases it may require for the user to provide it.

Our database example is formed by an array that stores the collections of *Movies*, *Directors*, and *MovieTheater*. We have supposed that initially a *Movie* object has five mandatory fields: *title*, *year*, *director*, *genre*, and *rating*, and an optional field *prizes*. The field *criticisms* was added later. Each *Director* object has the mandatory fields *name* and *directed-movies*, and the *actor-movies* optional field. In addition, the *films* field, intended to record the list of directed films, was renamed as *directed_films* as part of a database refactoring process. The *Criticism* embedded objects have the mandatory fields *content*, *journalist*, and *media*; the value of the media field can be a String or another *Media* embedded object that has the *name* an *url* fields, both are of String type. The *Prize* objects have the *year*, *event*, and *name* fields.

Note that while JSON syntax allows embedded objects to be explicitly represented, JSON does not provide any construct to explicitly express that a value is a reference to another object. Instead, references must be inferred from a data analysis. Some idioms have been therefore proposed to express references in NoSQL databases. For instance the *_ref* or *_id* suffix for the name of the fields whose value is a reference or the embedded object *{$ref: entity-*

name, $id: reference-id} (*DBRefs*, in the MongoDB terminology) that indicates the referenced value and the entity type. These idioms will have to be considered in our schema inference process (Chapter 5).

Figure 2.1 shows the nested structure of the *Movie* database object whose title is `"Truth"`, and Figure 2.7a shows the object tree of this *Movie* object.

2.6 Aggregate-Oriented Data Models

A limitation of the relational database paradigm is the lack of appropriate constructs to represent complex data [24]. Relationships among objects must be addressed through *joins* by means of foreign keys (i.e. references between tables). Object references and embedded objects are constructs specially conceived to represent complex data, but they are not part of the relational model. This limitation motivated the emergence of object-oriented databases and object-relational paradigms in early nineties. These two object database paradigms have been scarcely adopted by companies. Object-oriented databases have a small market niche (mainly CAD/CAM and multimedia applications) [32]. Object-relational databases extended the relational models with constructs intended to manage complex data. These extensions add complexity to databases and reduce the productivity of programmers, which motivated the lack of acceptance among developers of relational database applications [69]. As indicated above, NoSQL databases have emerged to meet the requirements of modern applications, by overcoming the limitations of existing database models.

Unlike object-oriented databases, aggregate objects are usually preferred to object references in the case of NoSQL databases, because the data is distributed through clusters to achieve scalability, and object references may involve contacting remote nodes. Thus, aggregate-orientation has been identified as a characteristic shared by the data models of the three most widely used NoSQL systems: key-value, document, and wide-column. They organize the storage in form of collections of key-value pairs in which the values can also be collections of key-value pairs. The “aggregate-oriented data model” term has been proposed to refer these three data models [109]. It is worth noting that in these models references among data are expressed in a similar way as foreign keys in relational databases, that is, the atomic value of a property (e.g. *director_id* in *movie*) matches a value in another property of a different object (e.g. *_id* in *director*), as shown in Figure 2.1.

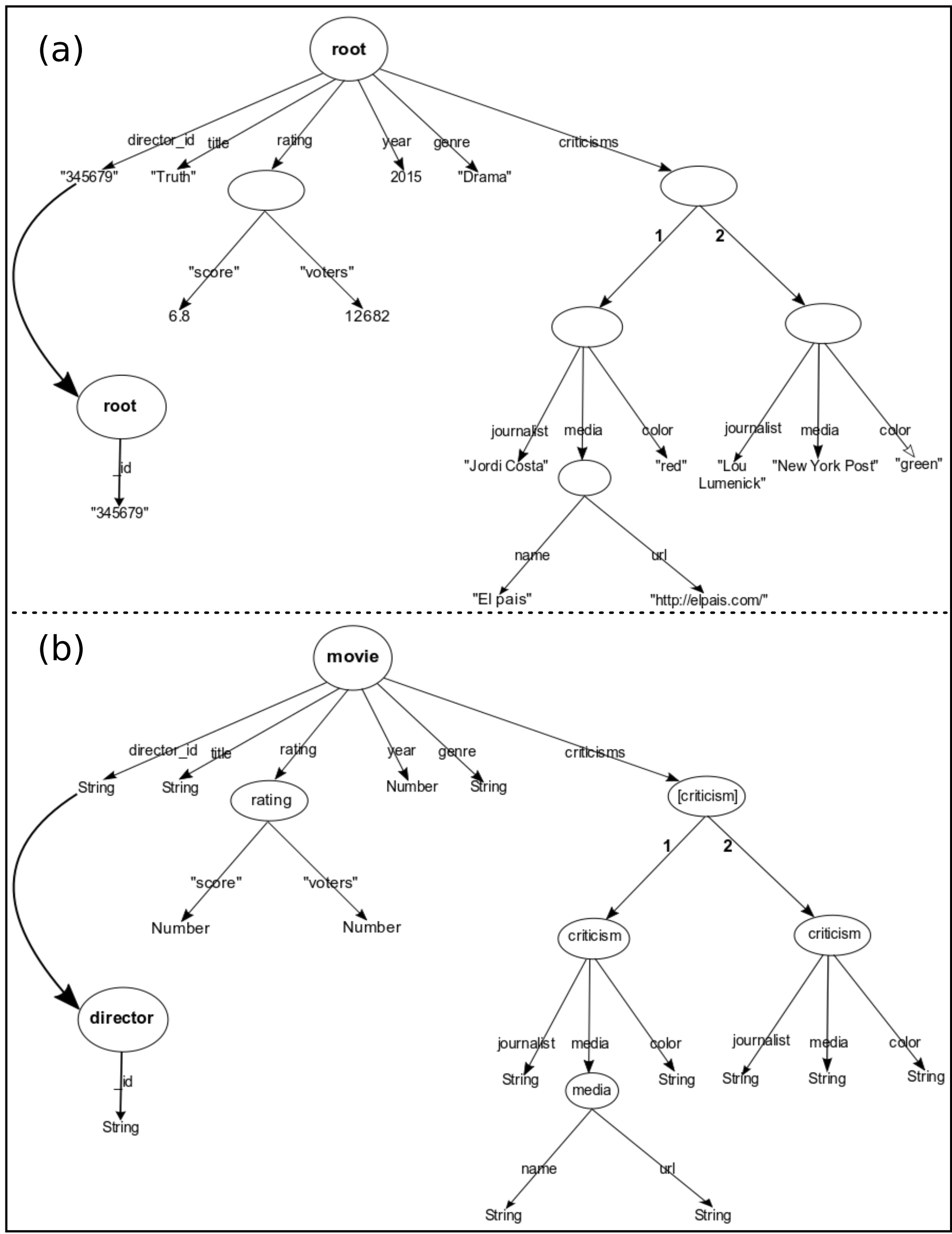

Figure 2.7: Object Tree and Type Tree for a Movie.

NoSQL databases store semi-structured data, and they are therefore characterized by the fact that explicit schemas do not have to be defined (*schemaless feature*), which provides them with greater flexibility. As explained above, non-uniform data can be stored,

```
{ "rows": [
  { "type": "movie",
    "title": "Citizen Kane",
    "year": 1941,
    "director_id": "123451",
    "genre": "Drama",
    "_id": "1",
    "prizes": [
      { "year": 1941,
        "event": "Oscar",
        "names": [
          "Best screenplay",
          "Best Writing"
        ]},
      { "year": 1941,
        "event": "New York Film Critics Circle Awards",
        "names": ["Best Screenplay"]
      },

      { "year": 1999,
        "event": "Village Voice Film Poll",
        "names":["Best Film of the Century"]
      },
      { "year": 1941,
        "event": "National Board of Review, USA",
        "names": [
          "Best Film",
          "Top Ten Films"
        ]}
    ],
    "criticisms": [
      { "journalist": "Roger Ebert",
        "media": "Chicago Sun-Times",
        "url": "http://chicago.suntimes.com/",
        "color": "green"
      },
      { "journalist": "Pablo Kurt",
        "media": "FILMAFFINITY",
        "url": "http://filmaffinity.com/es/main.html",
        "color": "green"
      },
      { "journalist": "Richard Brody",
        "media": "The New Yorker",
        "color": "green"
      }
    ]},
  { "type": "movie",
    "title": "The Man Who Would Be King",
    "year": 1975,
    "director_id": "928672",
    "genre": "Adventures",
    "_id": "2",
    "running_time": 129
  },
  { "_id": "3",
    "type": "movie",
    "title": "After hours",
    "year": 1985,
    "director_id": "907863",
    "genre": "Comedy",
    "prizes": [
      { "year": 1986,
        "event": "Independent Spirit Awards",
        "names": [
          "Best Best Feature",
          "Best Director"
        ]
      },
      { "year": 1986,
        "event": "Festival Cannes",
        "names": ["Best director"]
      }]
  },
  { "_id": "4",
    "type": "movie",
    "title": "Truth",
    "year": 2014,
    "director_id": "345679",
    "genre": "Drama",
    "rating":{
        "score": 6.8,
        "voters": 12682
    },
    "criticisms": [
      { "journalist": "Jordi Costa",
        "media": "El pais",
        "color": "red"
      },
      { "journalist": "Lou Lumenick",
        "media": "New York Post",
        "color": "green"
      }
    ]},
  { "type": "movie",
    "title": "Touch of Evil",
    "year": 1958,
    "writers": ["Orson Welles", "Whit Masterson"],
    "director_id": "123451",
    "genres": ["Thriller", "Classic"],
    "_id": "5"
  },
  { "name": "Orson Welles",
    "directed_movies": ["1", "5"],
    "actor_movies": ["1","5"],
    "type": "director",
    "_id": "123451"
  },
  { "type": "director",
    "directed_movies": ["4"],
    "name": "James Vanderbilt",
    "_id": "345679"
  },
  { "type": "director",
    "directed_movies": ["3"],
    "name": "Martin Scorsese",
    "_id": "907863"
  },
  { "type": "director",
    "directed_movies": ["2"],
    "name": "John Huston",
    "_id": "928672"
  },
  { "type": "movieTheater",
    "_id": "22",
    "name": "String",
    "city": "String",
    "country": "String"
  },
  { "type": "movieTheater",
    "_id": "23",
    "name": "Kinepolis",
    "city": "Madrid",
    "country": "Spain",
    "noOfRooms": 25
  }]}
```

Figure 2.8: Movie Database for the Running Example.

and the database evolution is facilitated. Figure 2.8 shows an example of NoSQL document database that stores JSON data on movies.

Several versions of an entity can be stored at the same time into a schemaless database due to the non-uniformity characteristic and the changes in the data structure when the database evolves. Database designers can consider initially some entity to have optional fields or fields that can have values of a set of types. Moreover, new versions of existing entities are created when the database evolves, that is, new fields may be added or removed, or the type of a existing field can be changed. Therefore, each entity will have one or more versions in the database schema. For example, in our *Movie* database example, there are 5 versions of the *Movie* entity, 2 versions of *Director*, and 2 versions of *Criticism*. Note that versions exist both for root and nested entities.

A schema of an aggregate-oriented data model for NoSQL database would be basically formed by a set of entities connected through two types of relationships: aggregation and reference. Each entity will have one or more properties or fields that are specified by its name and its data type. As indicated above, entities can be root or nested. A *root entity* is not nested to any other entity, and a *nested entity* is those embedded into a root or nested entity. However, the existence of entity versions means that each entity has as many schemas as it has versions, and then different kinds of schemas may be defined for aggregate-oriented data models, as discussed in Section 4.1. In that section, we will show how these schemas can be represented by labeled graphs. Figure 2.7 shows a kind of schema that is visualized as a entities tree (Figure 2.7b), which has been inferred from the Movie object that is also represented in form of a tree (Figure 2.7a).

In Chapter 6.1, we will present two notations for representing schemas for aggregate-based data models: a notation based on UML class-diagrams, and a notation specifically designed to visualize NoSQL versioned schemas. For instance, Figure 2.9 shows a possible visualization of the entity database schema for the Movies database, whose definition will be given in Section 4.1. Each entity schema is formed by joining the schemas of its entity versions, for example, *Movie* schema results of the union of the the *Movie* entity version schemas.

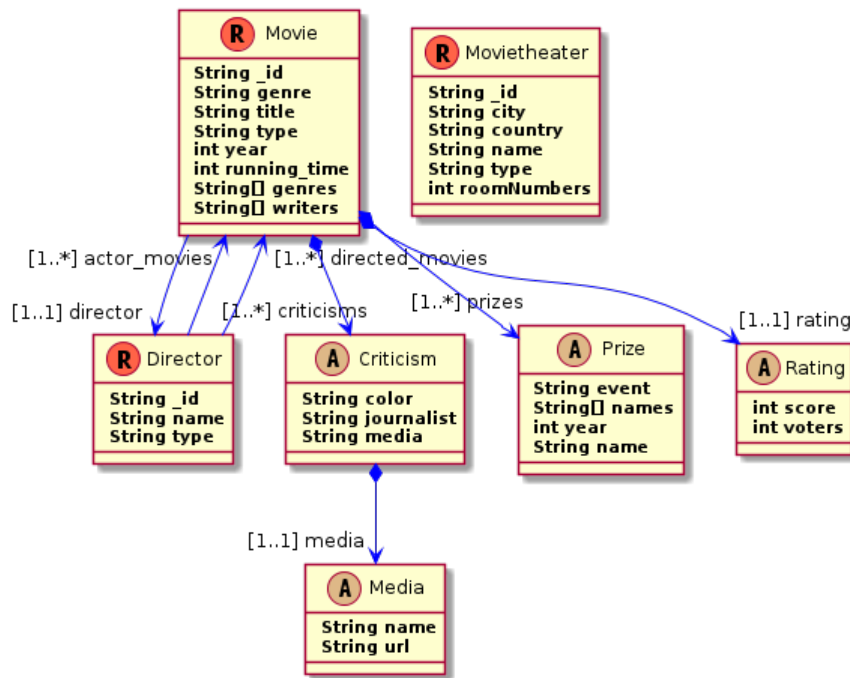

Figure 2.9: Object Tree and Type Tree for a Movie.

2.7 Object-Document Mappers

As commented above, in NoSQL database applications, the schema is in the mind of developers, so they have to devote a considerable effort to check that data managed in programs conform to the implicit schema. When database systems (e.g. relational systems) require the definition of a schema that specifies the structure of the stored data, a static checking assures that only data that fits the schema can be manipulated in application code, and mistakes made by developers in accessing data are statically spotted. However, schemaless databases entail developers to guarantee the correct access to data. This is an error-prone task, more so when the existence of several versions of each entity is possible. Therefore, some database utilities are emerging in order to alleviate this task. Object-NoSQL mappers are probably the more useful of these new tools. Like object-relational mappers, these mappers provide transparent persistence and perform a mapping between stored data and application objects. This requires that developers define a data schema, e.g. by using JSON [81], annotations [41], or a domain specific language [76]. Most of these mappers are for document

stores (Object-document mappers, ODM) because document-based databases (mainly MongoDB[†]) are the most widespread NoSQL systems. Mongoose [81] is the most widely used ODM, created for MongoDB and Javascript. However, ODMs for other languages, such as Morphia [83] for Java and Doctrine [41] for PHP, are also available.

It is worth noting that developers have two alternatives in building NoSQL database applications. They can work in a schemaless way or use an ODM mapper, by deciding on the trade-offs between flexibility and safety: they could prefer not having the restrictions posed by schemas or either avoid the data validation. ODM mappers can be a good choice when non-uniform types or custom fields are not needed, and the database schema will not change frequently. Next, we will introduce Mongoose as in this thesis we have developed code generators for these mappers.

Object-Document Mappers: Mongoose Mongoose is the *de facto* standard for defining schemas for MongoDB when writting Javascript applications. With Mongoose, database schemas can be defined as Javascript JSON objects, and then applications "can interact with MongoDB data in a structured and repeatable way" [61]. Since JSON is a subset of the object literal notation of JavaScript, the Mongoose schemas are really Javascript code. Schema definition objects are compiled into *models*. Such schemas are the key element of Mongoose, and other mechanisms are defined based on them, such as validators, discriminators, or index building. Instances of models map stored documents on which CRUD operations can be performed via the Mongoose API. An example with the creation of a *blog* schema can be seen in Figure 2.10.

2.8 MapReduce Operation

This thesis deals with NoSQL databases. Most NoSQL databases were born to overcome the problems with horizontal scalability of relational systems, so they offer algorithms for distributing, querying, and processing all the database data in a set of computation and storage nodes.

[†]Actually MongoDB uses BSON (Binary JSON), a variation of JSON with optimized binary storage and some added data types.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  }
});
```

Figure 2.10: Schema of a Blog Post in Mongoose.

One of the most important paradigms that allow that horizontal scalability is MapReduce [38, 39]. Its roots, as its name implies, comes from functional programming languages constructs: most functional languages provide equivalent *map* and *reduce* functions. *map* is a higher-order function that receives a list of values v_i and a function f and produces a list of resulting values, each one being the result of applying f to each original value v_i ($\{f(v_1), f(v_2), \ldots, f(v_n)\}$). The *reduce* function receives a list of values (in MapReduce it is usually the result of the *map* operation) and a function g, that, applied on the list, summarizes the values producing a single result.

The interest in these functional approaches is that it does not impose an order of evaluation, so effectively they can be performed in parallel. As the *map* function is the same to be applied to all data, if the data is distributed among several hosts, each of these hosts can perform the *map* function application in parallel.

The actual MapReduce paradigm is adapted to be applied to huge collections of data elements identified by a key, as most NoSQL databases are organized (see Section 2.4.) Thus, as described in [38], the types of the map and reduce functions are defined as:

- map (k1, v1) $\rightarrow$ list(k2,v2)
- reduce (k2, list(v2)) $\rightarrow$ list(v2)

The *map* function is applied to each data element, identified by the pair of key and value.

Each application of the map function generates zero or more pairs of key and value. Note that the domain of the produced keys and values may be different from the original data.

Then, a *suffle* process is performed by the run-time (usually the database) so that all the values produced under the same key are grouped, and thus, the *reduce* function will receive elements in the form of a key and a list of values. The function, then, reduces the initial set of data identified by this key to a set of small set of values, smaller than the initial set. This calculation involves generating statistics, counting values, etc.

Specifying processes using MapReduce requires a different perspective than the traditional imperative approaches or SQL based queries or processing. As an example of meaningful map and reduce functions, the literature traditionally shows how to calculate the set of different words in a text, and the number of times each word appears in the text [38].

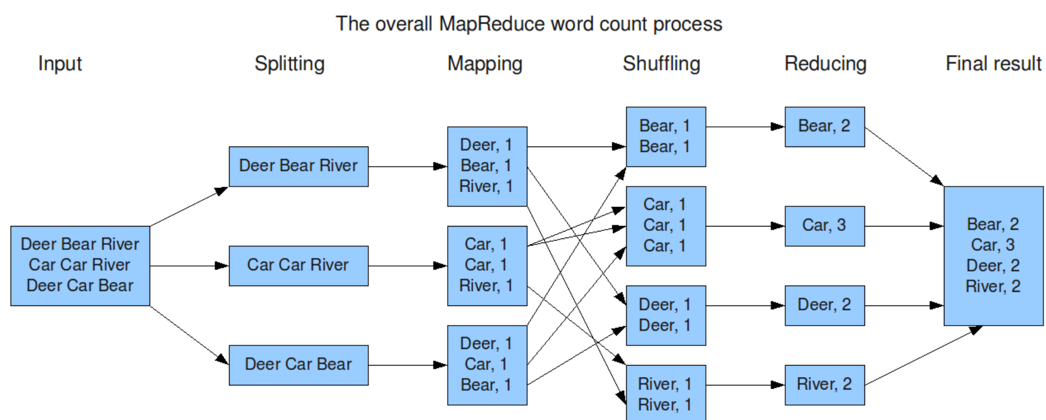

Figure 2.11: MapReduce word count example.

Figure 2.11 (taken from `https://cs.calvin.edu/courses/cs/374/exercises/12/lab/`) visually shows the process. In the figure, the initial input is taken from a complete document. The document is then split, which corresponds–in NoSQL database terminology–to different documents stored in the database. The map operation is performed in parallel to the different documents. In the word count example, the text of each document is separated in words, and, for each word, a key-value pair is produced in which the key is the word, and the value is 1, indicating that the map operation has found an occurrence of this particular word.

The shuffling operation connects all the elements under the same key (the same word) in a list. Each pairs then will contain the same key, and the list of values for this key. This can be simplified using the key and joining all the values in a list. This is exactly the arguments that the reduce operation receives. The results of the reduce operation are again word-count pairs. Joining all the results, the final result is obtained, containing all the different words and the count for each one.

In this thesis, MapReduce is used to achieve scalability, as the inference process is performed on all the objects of a database. The actual map and reduce operations used are explained in Section 5.1. For each document considered, its schema is constructed using an ASCII coding scheme. Then, the schema is used as a key, so all the objects that share the same schema are grouped by the shuffling process. Taking one of the group selects the archetype, and the list of archetypes generates a first hint of the different database types, used later for a finer inference.

2.9 Basis of Model-Driven Engineering

Model-Driven Software Engineering (MDSE or simply MDE) refers to an area of Software Engineering that addresses the systematic use of models to improve the software productivity and some aspects of software as maintainability and interoperability. Models can be used in the different stages of the software lifecycle to raise the abstraction level and automate development tasks. Although models have been used since early years of programming, MDE is a newly emerging discipline of Software Engineering. In November of 2000, OMG launches the MDA (Model-Driven Architecture) initiative [89]. MDA increased the interest in modeling by presenting a vision of the software development in which models are first-class citizens, like programs are. Although this idea was not really a novelty, MDA attracted the attention of the software community on the called model-driven development paradigm. Therefore, the emergence of MDA is usually considered the milestone that motivated the emergence of MDE.

Actually, MDA is only one of the existing MDE paradigms, but there are others, such as *Domain-Specific Development* [70, 116] and *Model-Driven Modernization* [93] which share the same four basic principles [16]: (i) models are used to represent aspects of a software system at some abstraction level; (ii) models are instances-of (or conform-to) metamodels;

(iii) model transformations provide automation in the software development process; and (iv) models can be expressed by means of modeling languages.

As noted in [16], there are four main scenarios in which MDE techniques can be applied: (i) building new software applications, (ii) software modernization (e.g. software re-engineering), (iii) use of models at runtime to represent the context or running software system, and (iv) integration of software tools. In this thesis, we have devised model-based solutions for tackling problems related to the two first scenarios: reverse engineering to infer database schemas in form of models, and developing database utilities from the extracted models.

Although MDE techniques are mainly used to create new software systems, they are also frequently used to automate software evolution tasks, for instance, to reverse engineer existing systems [51, 115, 95, 18]. Reverse engineering is based on code or data comprehension techniques. Chikofsky and Cross [28] define *reverse engineering* as *"the process of analyzing a subject system to i) identify the system's components and their interrelationships, and ii) create representations of the system in another form or at a higher level of abstraction"*. Reverse engineering can take advantage of MDE techniques. Metamodels provide a formalism to represent the knowledge harvested at a high-level of abstraction, and automation is facilitated by using model transformations. Therefore, we have devised an MDE solution to reverse engineer versioned schemas from aggregate-oriented NoSQL databases that we use to create database utilities. These utilities have been also developed by means of metamodels and model transformations.

Next, we shall shortly introduce the basic concepts of MDE.

2.9.1 Metamodeling

A *metamodel* is a model that describes the concepts and relationships of a certain domain. A metamodel is commonly defined by means of an object-oriented conceptual model expressed in a metamodeling language such as *Ecore* [114] or *MOF* [92]. A metamodeling language is in turn described by a model called *meta-metamodel*, therefore, a metamodel is an instance of a meta-metamodel and a model is an instance of a metamodel. The *four-level metamodelling architecture* is normally used to express the instantiation relationship between models and metamodels [16] as illustrated in Figure 2.12 that shows the four-level ar-

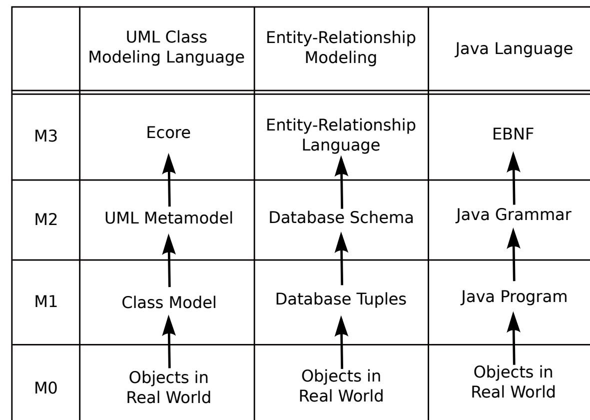

Figure 2.12: Four-Level Architecture for UML and ER Notations.

chitecture for the case of UML class models. A similar architecture can be defined for other technical spaces [74] as also shown in Figure 2.12 for relational databases technical space (schemas defined with the E/R language) and grammarware technical space (GLP programs conform to a grammar that is defined wit the EBNF language).

Metamodeling languages generally provide four main constructs to express metamodels: *classes* (normally referred to as metaclasses) for representing domain concepts; *attributes* for representing properties of a domain concept; *association relationships* (in particular aggregations and references) between pairs of classes to represent relationships between domain concepts; and *inheritance* between child metaclasses and their parent metaclasses for representing specialization between domain concepts. These four constructs are provided by the Ecore language [114]. Figure 2.13 shows the main elements of the Ecore that represent the mentioned four constructs. The metamodel concepts are represented by means of *EClass*; attributes are represented as *EAttribute*s, and *EReference*s allows us to represent relationships by using the *containment* boolean attribute to indicate its kind: aggregation (*containment=true*) and reference (*containment=false*). In the following chapters we will use metamodels with different purposes as representation of the implicit schema of a NoSQL database.

Usually, UML class diagrams are used to visually represent metamodels. For instance, Figure 2.14 shows a metamodel that represents the basic concepts of relational schemas. A *Schema* aggregates a set of *Table*s which, in turn, aggregate *Column*s and *Fkey*s (foreign

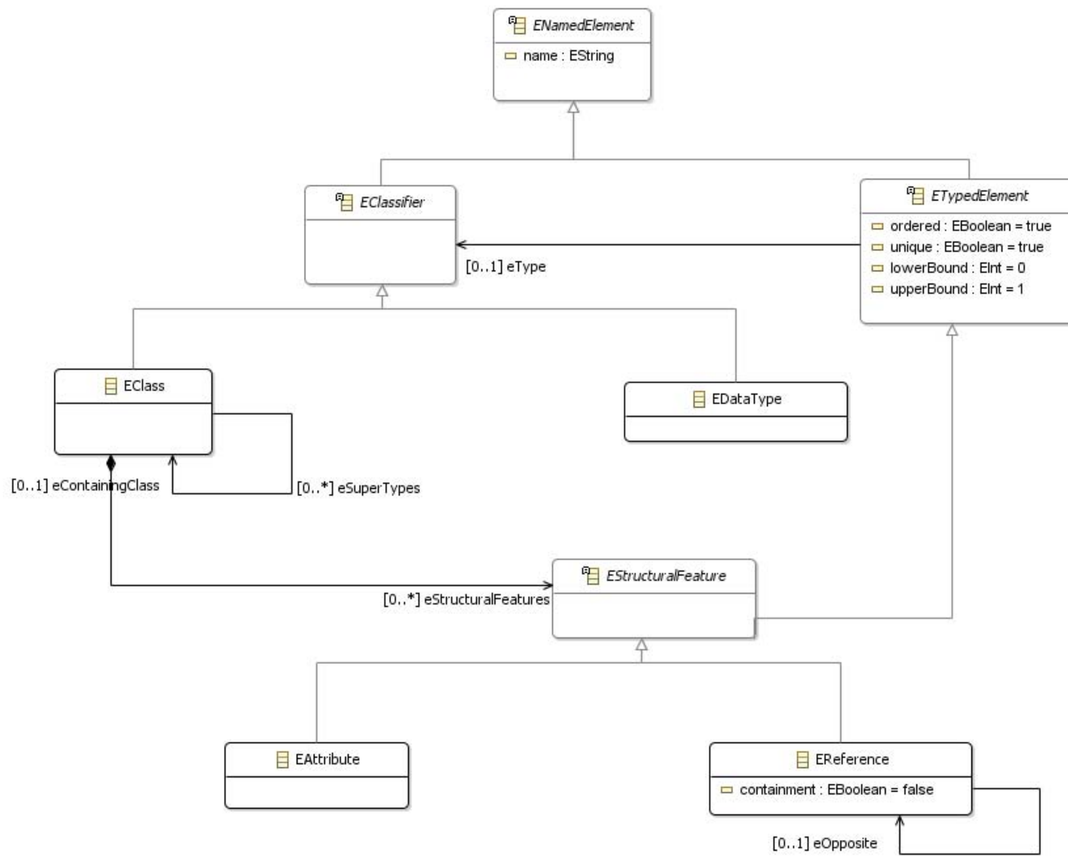

Figure 2.13: Main Elements of the Ecore Meta-Metamodel.

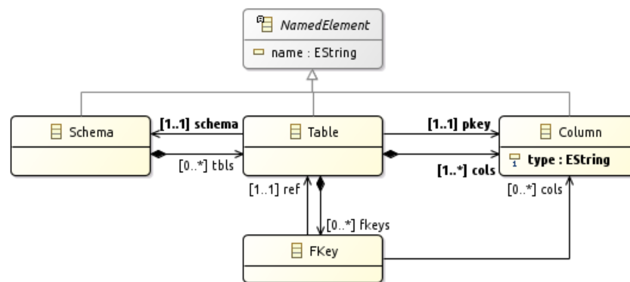

Figure 2.14: An Example Metamodel for Relational Schemas.

keys). Schemas, tables, and columns have a name, so that the corresponding classes inherit a *name* attribute from a *NamedElement* class. A column also has a *type* attribute of type String. *Table* has a reference to one or more *Column*s that form its primary key (*pkey*), and a *Fkey* has references to the columns that form it (*cols*) and the table to with it references (*ref*).

A metamodel must also include the well-formedness rules that constraint the set of valid models. Languages such as OCL (*Object Constraint Language*) [91, 120] are normally used to write these rules. Due to the limitations of UML diagrams to express complete and precise specifications, OCL was initially defined as a companion language for UML. OCL is a strongly-typed and declarative language that is widely used to write constraints and queries on class models. In MDE, OCL is especially used to create metamodels and writing model transformations (e.g. in the navigation of the source model). For instance, the following rule could be included in the above metamodel example to specify that an schema can not contain two tables with the same name.

```
context Schema
  inv: DifferentNamesForTables
    tbls-> forAll (t1 |
       tbls->forAll (t2 |
         t1.name = t2.name
            implies t1 = t2))
end
```

2.9.2 Domain-Specific Languages (DSLs)

Models (i.e. an instance of a metamodel) are created by means of textual or visual languages. When MDE solutions are developed for forward engineering scenarios these languages are needed to express the models from which target artifacts are generated. These languages are called *Domain-Specific Modeling Languages* (DSL) as they are designed to solve problems in a specific domain. However, sometimes modeling languages can be applied in any domain (*General-Purpose Modeling Languages)* [16]. Tools are available to define metamodel-based DSLs, that is, languages that allow creating models whose structure is determined by a metamodel. These tools facilitate the definition of a notation (concrete syntax) for the metamodel (abstract syntax), which can be textual or graphical (or a combination of both). This information (metamodel and notation) is used to automatically generate an editor and a model injector (i.e. a tool that create the instance of the metamodel from the visual or textual notation). The most widely used tools are Xtext [122] and MPS [84] for textual DSLs, and Sirius [7] and Metaedit [6] for graphical DSLs. In the case of textual DSLs, the tools provide a BNF-like notation to express the language grammar. For graphical DSLs, the tools provide a graphic editor to define the symbols of the metamodel elements and the tool palette of the editor generated.

DSLs have been used since the early years of programming, however, MDE has substantially increased the interest in them. Most MDE solutions involve the definition of one or more DSLs in order for users to create the required models. It is worth noting that when MDE is applied in reverse engineering scenarios, notations are not needed for the metamodels that represent the information gathered in that process if such information is not intended to be understood by users. In our case, the models obtained in the inference process are mainly intended for developers to help them to understand the database schema. As discussed in Chapter 6, one possible strategy to visualize the models inferred is to define a notation by using a DSL definition tool. However, we first visualized schemas by applying a generative approach to automatically generate diagrams from the models inferred, because this implementation requires less effort. Models have been directly manipulated by model transformations in order to achieve the diagramming of schemas and the implementation of some utilities such as validators or code for object-document mappers as described in Section 7.1. In our work, we have only created a DSL to express parameter models in generating

ODM schemas. This DSL is explained in Section 7.1.

In addition to *abstract syntax* and *concrete syntax*, a DSL has *semantics* as a third element. The semantics defines the behavior of the DSL; there are several approaches for defining it [71], but it is typically provided by building a translator (i.e., a compiler) to another language that already has a well-defined semantics (e.g., a programming language) or either an interpreter.

2.9.3 Model transformations

An MDE solution usually consists of a model transformation chain that generates the desired software artefacts from the source models. Three kinds of model transformations are commonly used: model-to-model (M2M), model-to-text (M2T) and text-to-model (T2M).

M2M transformations These transformations generate a target model from a source model by establishing mappings between the elements defined in their metamodels. One or more models can be the input and output of a M2M transformation. M2M transformations are used in a transformation chain as intermediate stages that reduce the semantic gap between the source and target representations.

The complexity of model transformations mainly depends on the abstraction level of the metamodels to which the models conform. The most frequently used M2M transformation languages (e.g., *QVT* [90], *ATL* [67], *ETL* [73]) have a hybrid nature since M2M transformations can be very complex to be expressed only by using declarative constructs [53]. These languages allow transformations to be imperatively implemented by using different techniques: i) imperative constructs can be used in declarative rules (e.g, ATL and ETL), ii) a declarative language is combined with an imperative one (e.g., QVT Relations and QVT operational), or iii) the language is designed as a DSL embedded into a general purpose language (e.g., *RubyTL* [34] into Ruby). Using model transformations to solve reverse engineering problems is an example of scenario where a high degree of processing of information is required and the complexity of transformations can become very high [105, 124]. Moreover, these reverse engineering tasks require the definition of intermediate data structures as graphs, trees, or tables, which are usually not supported by M2M transformation languages. The complexity of reverse engineering can be better tackled by using a model

management API (e.g. EMF) for a general purpose programming language (e.g. Java or Xtend) [115, 105, 124]. A survey on model transformation languages can be found in [35].

M2T transformations These transformations generate textual information (e.g. source code or XML documents) from an input model. M2T transformations produce the target artefacts at the last stage of the chain. *Mofscript* [43], *acceleo* [44], and the facilities supported by the *xtend* language [2] are some of the most widely used M2T model transformation languages.

T2M transformations These transformations (also called *injectors*) are used to extract models of the source artifacts of an existing system. They are mainly used in software modernization (e.g. in reverse engineering) to obtain the initial model to be reverse engineered. Hence, they are less frequently used than M2M and M2T. Among the tools for extracting models from code we remark the use of textual DSL definition tools as Xtext [122] or EMF-text [49]. A grammar is defined for the language of the input text, and the tool generates the model injector. Other tools used to inject models from source code are: *MoDisco* [17] that implements parsers (called *discoverers*) for Java and other languages, and the XML injector of the *Eclipse Modeling Framework (EMF)* [114] that obtains Ecore models from XML schemas.

In our work, we developed an initial prototype of our schema inference process by using (i) EMFText to create a model injector for JSON documents, and (ii) RubyTL to implement a m2m transformation that analyzed JSON models to obtain models representing the inferred schema. Finally, as explained in Chapter 5 the inference process was implemented in Java by using EMF API to build the extracted model.

2.10 Tooling, Frameworks, and Languages Used

NoSQL databases To show the validity of our approach in several databases and database types, we used two of the most common document-oriented NoSQL databases: MongoDB [80] and CouchDB [5]. We have started working also in HBase [57], but it requires further development, as HBase stores all the values as a byte blob. However, the approach has proved to be usable also in wide-column stores such as HBase.

As indicated in Section 2.7, we have automatically generated schemas for Mongoose.

Eclipse Modeling Framework and Ecore *Eclipse Modeling Framework* (*EMF*) [114] is the core infrastructure for the *Modeling Project* of the Eclipse platform. This project integrates a set of tools for applying MDE. EMF is currently the most widely used framework, and has significantly contributed to the fact that MDE can now be used in the academic and industry communities. EMF is composed of a metamodeling language called Ecore and the tooling needed for the creation and manipulation of Ecore (meta)models, which is usually referred also as EMF.

EMF supports two ways of manipulating models: code generation and dynamic models. Given a metamodel, EMF can generate a set of classes and interfaces which represent the metaclasses of the metamodel and can be instantiated to create models. This is called *generated EMF*. Instead of generating code, EMF also allows a model to be dynamically generated by means of the *dynamic EMF* API.

All the tools developed in this thesis use EMF/Eclipse as modeling framework.

Model transformations In our work, we have used dynamic EMF in Java code to implement the inference process. As indicated above, we have not used a M2M transformation language as they are not appropriate for complex reverse engineering processes. With regards to M2T transformations, we firstly used the MOFScript language [43] but we migrated our code to Xtend [2] to achieve a better interoperability and support. Recently, MOFScript has been discontinued and this supports that our decision was appropriate.

Xtend [2] is a dialect of Java that compiles to Java 5 and is available as a Eclipse plugin that is integrated into EMF. This language was devised with the aim of offering to the Xtext community a powerful language to provide semantics (i.e. M2M and M2T transformations) to the DSLs built with Xtext. In fact, Xtend was implemented by using Xtext. Xtend was presented as a Java modernized with advanced features as lambda expressions, extension methods, and type inference. Most of these novelty features are currently supported by Java 8. However, Xtend provides an interesting mechanism that is not provided by Java 8: *a template definition mechanism* which allows M2T transformations to be written. This mechanism offers language constructs to easily navigate through the input models and the text generated can be specified as the input model is traversed.

Other MDE tools Xtext [122] is a very widespread DSL definition tool that has achieved a high level of maturity along last eight years. Xtext together Xtend form a powerful environment to build DSLs. Xtext offers a simple BNF-like language to define the concrete syntax of a DSL. From a grammar specification, Xtext generates the DSL metamodel, an editor and a model injector. We have used Xtext to create the *ODM Parameter DSL* described in Section 7.1.

PlantUML [96] is a textual DSL aimed to draw UML diagram. The notation is simple and easy to learn and use. The PlantUML engine generates

DOT code and use Graphviz [55] for drawing UML diagrams. We have used PlantUML to visualize the NoSQL schemas inferred as models in form of UML class diagrams, as explained in Chapter 6.

We ourselves feel that what we are doing is just a drop in the ocean. But the ocean would be less because of that missing drop.

Mother Teresa of Calcuta

3 State of the Art

Database management systems (e.g. Relational, Object-oriented or NoSQL) are a key element of software applications. *Data Engineering* is the Computer Science discipline concerned with the principles, techniques, methods and tools to support the data management in the software development. Some of the main topics of Data Engineering are Data Evolution (e.g. data migration), Data Reverse Engineering, Data Integration, and Data Tooling. Data Engineering has been mainly focused on relational data so far, although interest is shifting towards NoSQL databases. Therefore, NoSQL Data Engineering is an emerging area which is increasingly attracting the attention of industry and academia. In our research work, we have tackled the inference of implicit schemas in NoSQL databases and the usage of the inferred schemas to develop database utilities. Therefore, this thesis contributes to the NoSQL Data Engineering area with one of the first works on reverse engineering as well as the automation of some tasks by means of utilities generated from schemas.

We have organized the related work analyzed in three main categories: (i) NoSQL schema inference approaches, (ii) NoSQL schema representations, and (iii) development of database utilities. In the first category we have also considered the works related to the extraction of XML and JSON schemas. In the third one, we have focused on schemas visualization tooling. We end this chapter with a final discussion that contrast the related work with the pro-

posals presented in this thesis.

3.1 NoSQL Schema Inference

The schema extraction for JSON-based technologies and applications is gaining attention as JSON is becoming a *de facto* standard in information interchange. Also, JSON-based NoSQL stores are emerging. This research effort is related to the works published over the years on schema inference and schema versioning for semi-structured data, specially XML documents. The works most closely related to the NoSQL schema extraction approach devised in this thesis are [72] and [119]. Next we will discuss in detail these two works. Moreover, we will also comment on other relevant works on schema inference from NoSQL databases, XML, and JSON.

3.1.1 Meike Klettke et al.

An algorithm to extract schemas from aggregate-oriented NoSQL databases is presented in the work of Klettke et al. [72]. This algorithm adapts strategies proposed for extracting XML DTDs [79] to JSON documents. The authors use JSON Schema to represent the output. A JSON Schema is extracted from a collection of JSON documents. Figure 3.1 shows the schema obtained for the *Movie* collection of our running example.

Noting that the JSON schema shown in Figure 3.1 for the Movie collection corresponds to the notion of *union object schema* given in Section 4.1, only that aggregated types are labeled as *Object* and arrays of objects as *array*. Therefore, this work does not discover what entities and entity versions exist in the database and, therefore, any of the kinds of versioned schemas defined in Section 4.1 have been considered. Instead, the approach of Klettke et al. only obtains a schema for each collection, which is called *type*, that is defined as the union of the object schemas of each version. That is, an entity schema is formed by the union of all the fields of the versions that exist for that entity. For instance, the schema obtained for the *media* field is *[{"type":"string"},{"type":"Object"}]*, according to the variation of this field in our database example. References are also identified. For instance the field *director_id* references to *Director* entities, but this is not specified in the JSON schema obtained.

Given a collection of documents as input, the algorithm of Klettke et al. works in four steps as shown in Figure 3.2. In a first step, a selection of the documents to be analyzed is

```
{
    "type": "object",
    "properties": {
        "_id": {
            "type": "integer"
        },
        "title": {
            "type": "string"
        },
        "year": {
            "type": "string"
        },
        "director_id": {
            "type": "string"
        },
        "genre": {
            "type": "string"
        },
        "running_time": {
            "type": "integer"
        },
        "rating": {
            "type": "object",
            "properties": {
                "score": {
                    "type": "integer"
                },
                "voters": {
                    "type": "integer"
                }
            },
            "required": [
                "score",
                "voters"
            ]
        },
        "prizes": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "year": {
                        "type": "integer"
                    },
                    "event": {
                        "type": "string"
                    },
                    "names": {
                        "type": "array",
                        "items": {
                            "type": "string"
                        }
                    }
                },
                "required": [
                    "year",
                    "event",
                    "names"
                ]
            }
        },
        "criticisms": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "journalist": {
                        "type": "string"
                    },
                    "media": {
                        "anyOf": [
                            {
                                "type": "string"
                            },
                            {
                                "type": "object",
                                "properties": "..."
                            }
                        ]
                    },
                    "color": {
                        "type": "string"
                    }
                },
                "required": [
                    "journalist",
                    "media",
                    "color"
                ]
            }
        }
    },
    "required": [
        "_id",
        "title",
        "year",
        "director_id",
        "genre"
    ]
}
```

Figure 3.1: JSON Schema of our database example.

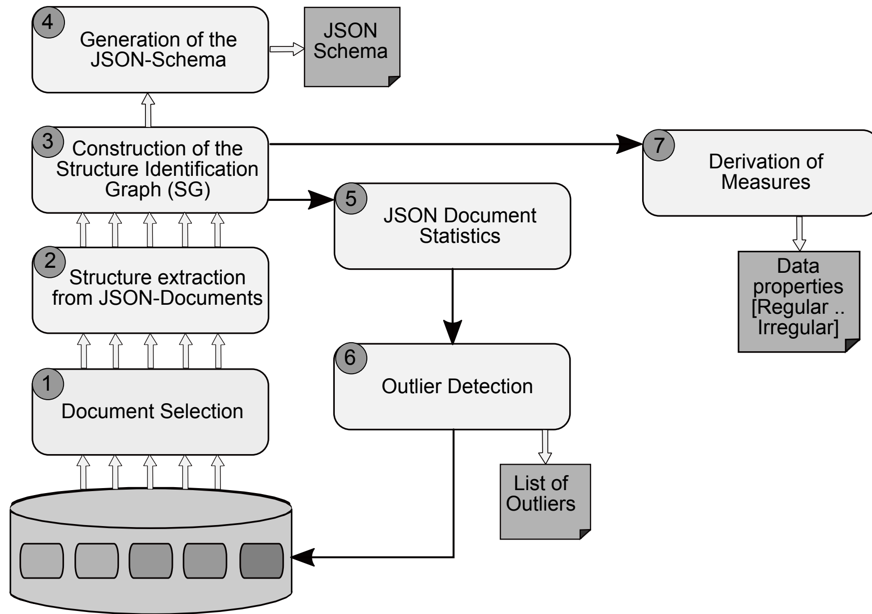

Figure 3.2: Four steps of the approach of Klettke et al. (extracted from [72]).

performed: either the entire collection or a subset of it. When this second option is chosen, the subset is formed by considering properties as version number, timestamp, date, or a particular *split* attribute. Next, the algorithm parses selected JSON documents to construct the *Structure Identification Graph* (*SG*) that represents the structure (i.e. the type or entity schema) of a document (i.e. a root object) of the collection. Figure 3.3 shows the SG built for the Movie_2 and Movie_4 objects of our database example. Nodes correspond to JSON values and edges capture the hierarchical structure. Nodes and edges are labeled with information that indicate which JSON document the property belongs to. This information consists of lists of identifiers of all the documents that include a particular property. Each property is defined by the path of keys formed by the ordered list of keys of the objects parents, which starts with the key of the root object. Moreover, the *SG* nodes store the data type of each field. Finally, the *SG* graph is traversed to generate the JSON schema. The information stored into the *SG* nodes allows to distinguish when a field of an root or embedded object is required or optional.

The JSON schema generated is used to calculate statistics and metrics, and finding outliers in the stored data. This information is generated in the process of construction of the

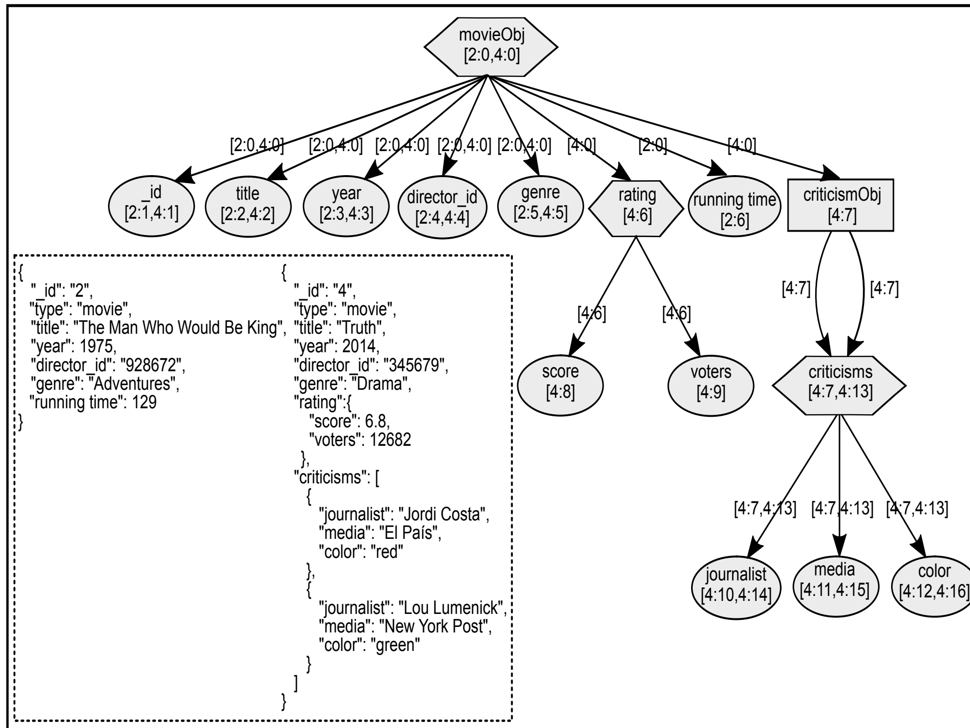

Figure 3.3: Structure Identification Graph for a Movie.

SG graph. An example of statistics offered is the percentage of documents including a particular field. These percentages can be used to find outliers as missing properties (i.e. those properties occurring in nearly all the documents) or additional properties (i.e. those properties rarely occurring). Among the metrics, the degree of coverage for documents is proposed in order to measure the structural homogeneity of a collection. The authors also suggest as further work some database utilities similar to those developed in this thesis, such as validators and objects mapper classes.

The notion of NoSQL schema presented in our work is more expressive than the JSON schemas in the standard, since NoSQL schemas contain aggregation and reference relationships between entities, and also entity versions are extracted and represented. Our work therefore differs from this approach in several essential aspects: i) the algorithm of Klettke et al. identifies the required and optional properties, but object version schemas are not obtained; ii) schemas involve an *Object* type, and reference and aggregation relations are not considered; iii) they do not specify how to cope with huge amounts of data; and iv) we ob-

tain a model that conforms to a metamodel, instead of a JSON Schema.

3.1.2 Lanjung Wang et al.

The authors present a schema management framework for NoSQL document stores [119]. To our knowledge, this is the only proposal that, like our work, deals with entity versions. The main elements of this framework are shown in Figure 3.4: (i) an algorithm for discovering and extracting schemas; (ii) an approach for querying extracted schemas, and (iii) a format to present all the schema versions of an entity as a single *approximate* schema, in order to make it easier for users to understand the structure of stored documents.

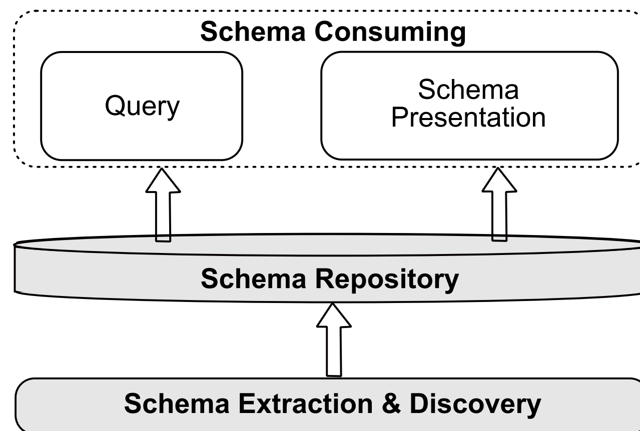

Figure 3.4: Schema management framework of Wang et al. (extracted from [119]).

Three main challenges have been identified in building this framework. First of them has to do with the existence of different schema versions for the entities stored into a collection. Moreover, the authors consider that a collection can have tens of thousands of versions in some real-world scenarios. For instance, they have identified 21,302 versions of the *Company* entity in the 24,367 documents of the DBpedia database. This database is one of four analyzed case studies. The first challenge to be tackled is therefore the efficiency of the process of schema discovering and extracting, which must also support efficient online updates. The other two challenges are related to queries in the discovered schema repository. The authors have defined a SQL-like API to perform queries over the schema repository. They have defined only two basic queries as are checking the existence of a particular schema or subschema. Since the authors assume the existence of a very large number of schema versions

for an entity, then they consider that a single schema should be returned, rather than the collection of all the existing versions, when data scientists perform queries over the schema repository in advanced data exploration scenarios. They discarded the *union* and *intersection* of schemas because the first operation could generate schemas with a large number of properties, whereas the intersection could originate schemas with a very reduced number of properties. For instance, in the case of the *Company* entity of DBpedia, the intersection is only one property, whereas there are 1,648 properties for the union.

To achieve an efficient schema discovering algorithm, the authors have devised a data structure called *eSiBu-Tree* (encoded *Schema in Bucket Tree*) to represent schemas both in memory during the inference process and on disk when they are persisted. The queries are efficiently performed by using this data structure. To summarize schemas, the notion of *skeleton schema* is proposed. The eSiBu-Tree and the skeleton notion are the two main contributions of the work of Wang et al., which support the schema management framework proposed. Next, we will give some details on these two concepts.

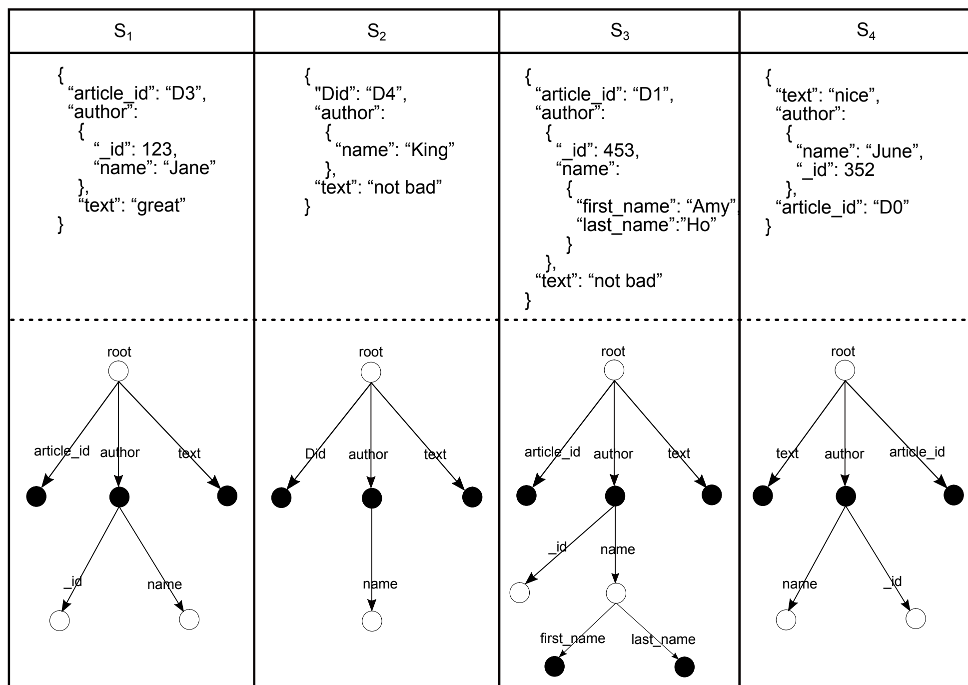

Figure 3.5: Documents stored in *Article* collection (extracted from [119]).

The design of the eSiBu-Tree data structure was due to the inefficiency observed to group

equivalent schemas by means of a Canonical Form (CF)-based method, in particular the method for generating Depth-First Canonical Form [27]. This method analyzes all the objects of a collection and groups the fields by levels as illustrated in the example of Figure 3.6 that shows the generation of the Canonical Form for the four JSON documents shown in Figure 3.5. These documents are stored in a collection for an *Article* entity which has three properties: *identifier*, *text*, and *author*. The number of levels is determined by the maximum level of nesting in the documents. In this example, there are 4 levels that correspond to *root.author.name.firstName* and *root.author.name.lastName*. Figure 3.6 illustrates how iteratively works the CF-based algorithm. An array of label-code pairs is generated for each level. This array is called *code map array*. Given a nesting level in the hierarchical structure of the documents stored into a collection, a code map array contains a pair for each field with different type; in the example the array for the level 2 includes pairs for the following fields: *article_id*, *text* and *Did* of String type; and three pairs for the *author* field that corresponds to the documents S1, S2, and S3 (this field has the same type in S4 and S1). Each pair of a code map array has a code that is assigned as follows: 1 is assigned to the first field found for that level, and this value is incremented by one for each new pair added to the array. The label of a field of Object type results of concatenating the name of the field with the ordered codes of its children fields, which are part of the map code array of the next level. Therefore, a sort of codes is needed to form the labels of these fields. In the case of fields of primitive type, the label is only formed by the name of the field.

The performance of the CF-based algorithm for grouping schemas depends on the number of sorts of codes performed. Therefore, Wang et al. defined the eSiBu-Tree data structure which allows define a divide-and-conquer algorithm that reduces the number of sort by half. This algorithm also provides better performance than the CF-based algorithm for querying schemas and obtaining the schema skeleton. Figure 3.7 shows the eSiBu-Tree generated as result for documents in Figure 3.5. Each node or bucket belongs to a single schema, and a schema is represented by the path going from the root bucket to a bucket that represent its last level of nesting. A bucket is formed by four elements: (i) an identifier that is a ordered sequence of codes of pairs in the parent bucket; (ii) an array of label-code pairs like that described for the CF-based algorithm, but a bucket only contains the pairs of a particular schema, instead of all pairs of all the schemas for a determined nesting level (level 2 in

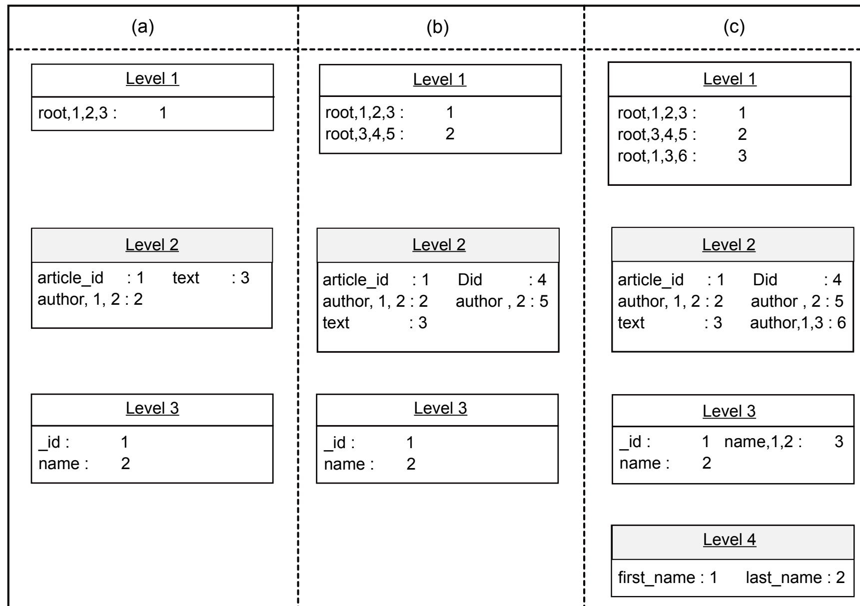

Figure 3.6: Canonical form generated for documents in Figure 3.5 (extracted from [119]).

Figure 3.6 includes three pairs for authors, but a bucket only contains a pair for each field included), (iii) a flag that indicated whether a bucket is the end of the path representing a particular schema; and (iv) a list of children buckets. In Figure 3.7, the tree (c) shows the eSiBu-Tree that is finally generated. The root of this tree only includes an array of label-code pairs for all the fields in root documents. This root bucket has two children bucket because two potential schemas can be identified at level 1 ({*article*, *_id*, *text* and *author*} and {*Did*, *text*, and *author*}). The {ID:1,2,3} bucket for the former schema includes pairs for the *_id* and *name* fields of *author*, and has a list of two buckets whose flag is *true*, these two buckets end the path of buckets for the schemas of the documents S1 (and S4) and S3 in Figure 3.5. The {ID:2,3,4} bucket for the second schema includes a pair for the *name* field, and a list of only a children which ends the path of buckets for the schema of the S2 document.

A *skeleton* is the smallest attribute set that is "enough" to characterize to the schema of the entity associated to a collection. The authors have defined some quality criteria that guide the skeleton construction process to find out the highest quality attribute set. They have defined an algorithm based on the eSiBu-Tree generated in the schema extraction phase for the skeleton construction.

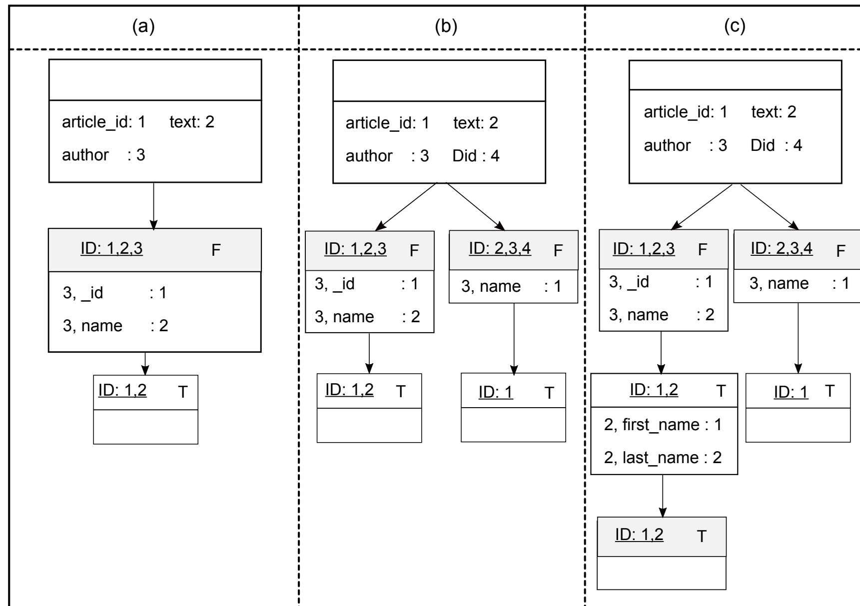

Figure 3.7: eSiBu-Tree generated for documents in Figure 3.5 (extracted from [119]).

Actually, schema extraction from semi-structured data has been already addressed in several works as [9], as well as the problem of finding the minimal perfect or approximate type of a object graph (i.e. a concise an accurate summary of a give data graph). Wang et al. note that the main contributions of its work are (i) tackling the efficiency issue, and (ii) the identification and usage of quality criteria to construct the skeleton [119].

3.1.3 MongoDB Schema

MongoDB-Schema [108] is an early prototype of a tool whose purpose is to infer schemas from JSON objects and MongoDB collections. Given a set of objects of the same collection, the inference algorithm extracts an schema that is similar to that obtained with the approach of Klettke et al. [72], that is, the union of the object schemas of the different versions of an entity. When a field belongs to more than one entity version, its type is the union of the types encountered for this field. Moreover, metadata is added to each field in the root and embedded objects in form of a key/value pair. For instance "type" indicates the object type (e.g. *Number*, *String*, or *Boolean*), "count" indicates the number of objects that contains a field, "probability" for a field indicates the percentage of objects that have it, "probability"

for a type indicates the percentage of objects that have the field/type pair. The schema can also collect the values sample for each field/type pair. Figure 3.8 shows the structure of a schema in MongoDB-Schema. An entity schema is formed by a collection of fields, and each field has one or more types that can be *ConstantType*, *PrimitiveType*, *Array* and *Document*. MongoDB-Schema supports the set of BSON types.

```
{
  "count": 4,                    // parsed 4 documents
  "fields": [                    // an array of Field objects
    {
      "name": "_id",
      "count": 4,                // 4 documents counted with _id
      "type": "Number",          // the type of _id is `Number`
      "probability": 1,          // all documents had an _id field
      "has_duplicates": false,   // therefore no duplicates
      "types": [                 // an array of Type objects
        {
          "name": "Number",      // name of the type
          "count": 4,            // 4 numbers counted
          "probability": 1,
          "unique": 4,
          "values": [            // array of encountered values
            1,
            2,
            3,
            4
          ]
        }
      ]
    },
    {
      "name": "a",
      "count": 3,                // only 3 documents with field `a` counted
      "probability": 0.75,       // hence probability 0.75
      "type": [                  // found these types
        "Boolean",
        "String",
        "Number",
        "Undefined"              // for convenience, we treat Undefined as its own type
      ],
...
```

Figure 3.8: Partial Structure of a Entity Schema in MongoDB-Schema (extracted from [108]).

The schema example in Figure 3.8 is taken from the documentation of MongoDB-Schema [108]. The *_id* field of *Number* type is encountered in the four parsed documents, but the *a* field is only encountered in three documents and has a different type in each document: *Boolean*, *String*, and *Number*. The meaning of metadata considered can be easily understood from the comments added. The set of built-in types can be extended with custom types by adding a *detector function* for values of that type, and it is also possible to override the function that identifies values of a built-in type.

3.1.4 SQL query engine for structured data sources

Some NoSQL database tools have recently emerged which offer functionality that requires discovering the schema of the data stored. Spark SQL [123] and Drill [42] query engines are examples of such tools. Spark SQL Apache provides uniform access to a variety of structured data sources. In Spark SQL, a schema is described as a set of Scala algebraic types and can be inferred for a given set of JSON objects. Spark addresses object versions by means of "sum types", that is, creating types that contain all the properties in all the objects of an entity type, allowing them to be *null* in the objects created or received. Noting that this notion of "sum types" corresponds to the concept of "union object schema" defined in Section 4.1. As for conflicting types, it generalizes to a String type, that is able to represent any value. This may allow these conflicting types to be addressed without crashing, but it does not offer any guarantee regarding the consistency of the data.

Apache Drill [42] is a SQL query engine for Hadoop, NoSQL and Cloud storage. It dynamically discovers the schema during the processing of a query, but it cannot cope with conflicting objects (those that do not comply with the schema). Also, the discovered schema is just used for the purposes of Drill, and cannot be reused by other applications.

3.1.5 JSON Discoverer

A MDE-based approach to infer JSON schemas from JSON-based Web APIs is proposed in [64, 65]. This work is motivated by the fact that the integration or reuse of Web APIs requires an understanding of the data model behind them. However, the schemaless characteristic of JSON makes it difficult for developers to gain this understanding. Therefore, the authors propose a solution aimed to discover and visualize implicit schemas in JSON data. A three-step process is performed to discover the domain model of the services. Firstly, the JSON data for a service is injected into models which conforms to a JSON metamodel. The JSON model injector and JSON metamodel has been generated by defining a JSON grammar in Xtext [122]. In the second step, a mapping between the JSON metamodel and the Ecore meta-metamodel is established in order to transform the JSON model into a domain model. JSON objects and pairs are transformed into Ecore classes and references. Each class of the domain model generated corresponds to the notion of entity union schema that was defined in Section 4.1. The resolution of conflicting types is addressed by generalizing to

the String type just like in Spark SQL. Finally, the domain models obtained for each service are integrated by superposing the common classes. Therefore, the domain model generated would correspond to the notion of entity database schema defined in Section 4.1 but excluding references between entities. A tool that implements this approach can be online executed in [40]. This tool is called JSON Discoverer and offers three functionalities: (i) *simple discovery* that discovers the schema for JSON documents of a single service; (ii) *advanced discovery* that provides the global schemas of a Web API by integrating the schemas discovered for each service; and (iii) *Composition* that composes the schemas of several Web APIs. Schemas are drawn as UML class diagrams: entities and properties are represented as classes and attributes, respectively; and relationships between entities are represented as composite associations. Web API composition are represented as UML sequence diagrams.

This schema discovering work of Canovas and Cabot [64, 65] is close to our approach but there are some significant differences between them as indicated below. Unlike the works previously analyzed, this inference process extracts entity domain models rather union object schemas, which requires discovering and extracting the involved aggregation relationships. However, the existence of data versions (i.e. versioned schemas) is not addressed, and the references between objects are not discovered. This can be easily checked by trying visualize the schema for our Movie database example in [40]. Another remarkable difference of the inference process has to do with the strategy applied to resolve conflicting types in building a entity as the union of the schemas of the different versions. We extracted the union of all the types identified, instead of generalizing to the String type. It is remarkable that our work focused on NoSQL databases, whereas JSON Discoverer is a solution intended to developers that manage Web APIs. Therefore, we had to take into account some specificities of databases, such as the very large of documents that can exist in a collection. Thus, a JSON model injection is not feasible in our solution, instead we have used a map-reduce operation as explained in Chapter 5. Moreover, we have extracted models that are instances of a metamodel that represents NoSQL schemas, instead of generating domain models that are instances of the Ecore meta-metamodel. Finally, it is worth to note that the JSON-to-Ecore mapping defined in the work of Cánovas and Cabot is similar to the one we have defined for obtaining a visual representation of the global schema, which will be explained in Section 6.2. We have converted the inferred schema models into Ecore

elements in order to take advantage of the visualization utility for metamodels provided in Eclipse/EMF. However, in our case, the mapping is more direct.

3.1.6 ExSchema

ExSchema [23, 22] is a tool aimed to discover schemas for NOSQL stores by applying a static analysis of the source code of applications that use data store management APIs. The tool supports different kinds of data stores: document, column family, graph and relational. ExSchema has been implemented as an Eclipse plugin, and can be integrated with Git repositories to continuously analyze the application code in order to evolve the schema when is needed. The schema discovering process is performed in two steps. Firstly, the Java code of database applications is analyzed and the schema discovered is represented by means of elements of a meta-layer (i.e. a metamodel) based on the SOS meta-layer proposed to offer a common interface for accessing a NoSQL stores [11] which is described in more detail later in this chapter.

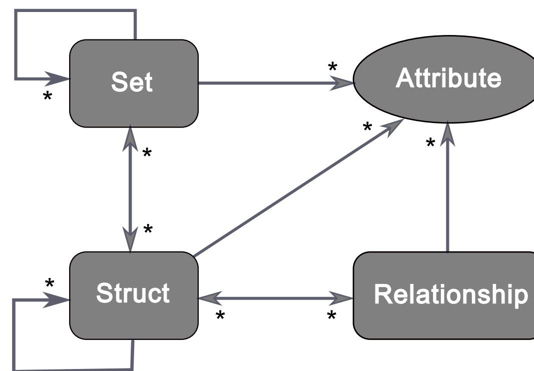

Figure 3.9: ExSchema Metalayer (extracted from [23]).

SOS has been extended with a new *Relationship* element to support graph stores, as showed in Figure 3.9. The discovered schema is visualized as a PDF image, and a Spring Roo [101] is generated to modify the original source code in order to work with the schema. As can be seen in Figure 3.10 (extracted from [22]), ExSchema visualizes schemas with a large number of visual elements, which makes it difficult to understand what are entities and which are the relationships among them.

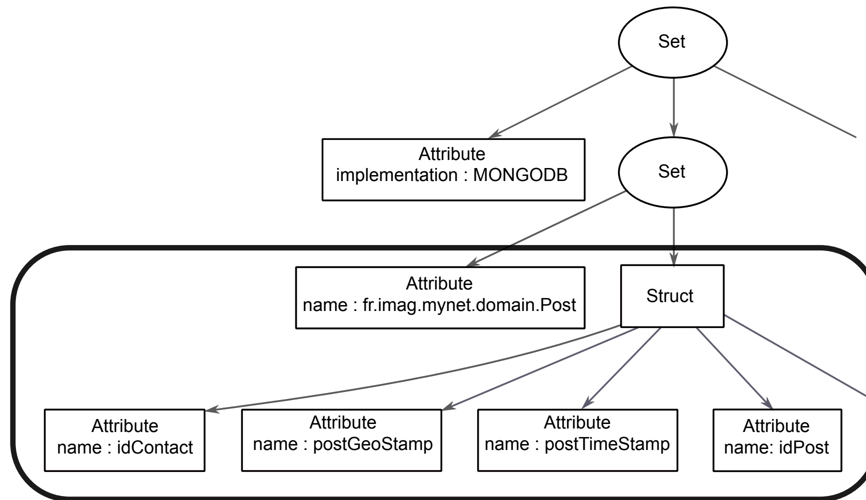

Figure 3.10: Generated diagram by the ExSchema tool for a MongoDB collection (extracted from [22]).

3.1.7 Early works on schema extraction for semi-structured data

At the end of the nineties, structuring or typing semi-structured data was a novel topic and an area of much research activity [8, 9, 20]. Although one of the main attraction of semi-structured data is being schemaless, several utilities of discovering the data structure was identified as: to optimize query evaluation, to facilitate the data integration, to improve storage, to construct indexes or to describe the database content to users [9]. Then, the main concern was to formally define the concept of data type (i.e. schema) for semi-structured data. Schemas are defined by means of some kind of graph-like structure as *edge labeled graphs* [20] based on *Object Exchange Model* (OEM) [8]. First-order logic (Datalog) and simulation were formalisms used to formally describe the notion of schema [9]. The schema discovery was mainly addressed as the problem of finding the most specific schema or approximate type for a set of data of the same entity. A clustering-based algorithm for approximate typing of semi-structured data was presented in [86]. In [118], an algorithm to find the typical type of a majority of objects in a collections is described. These authors tested the algorithm on the IMDb, in particular they chose 100 top movies from the list of 250 most voted movies. Then they converted HTML documents of these movies into OEM models in order to execute its algorithm, and obtained three more frequent object schemas, which were called schema patterns, which described to a 22%, 17%, and 16%.

In our work, we are not interested in obtaining an approximate schema for an entity, but we discover and extract all the entity versions. In order to present a summary schema, we obtain the entity union schema. Moreover, our discovering process outputs a model that conforms to a metamodel in order to facilitate the development of database utilities.

XML Schema Extraction The eXtensible Markup Language (XML) has been the predominant format for data exchange on the Web. Because the definition of a schema (e.g. a DTD or XML Schema) is not mandatory to create XML documents, the XML schema extraction has received a great attention from the database community. The process of extracting an XML schema (usually a DTD) consists of two main steps: discovering the hierarchical structure of the documents and transforming it into the schema representation. In [79], the authors present a tool able to infer the DTD of a set of XML documents. The approach applied to build this tool, which is called *dtd-miner2000*, works as follows. Each XML document is Firstly represented in form of n-ary tree, then the overall structure of all the structurally similar document trees is represented in a spanning graph, and finally some heuristics are applied to generate a DTD from the information in the spanning graph. The Structure Identification Graph (SG) used in the approach of Klettke et al. described above is based on this spanning graph data structure. In [14], the problem of inferring an XML schema is considered a problem that "basically reduces to learning concise regular expressions from positive examples strings", and several algorithms are described in detail. An efficient algorithm is presented in [58] which is based on the ideas exposed in [78], where the form of the regular expressions is restricted and some heuristics are proposed. In [66] a strategy to infer schemas from heterogeneous XML databases is presented. The schema is provided as a Schema Extended Context-Free Grammar, and the different versions are integrated into a single grammar which is mapped to a relational database schema. An algorithm to infer a succinct type (i.e. a schema) for a JSON dataset is proposed in [31], which works in two phases. In the first phase a Map-Reduce operation is applied. The map operation infers the type of each JSON object and generates a pair whose key is the inferred type and the value is 1. The reduce operation counts the number of objects of each type, that is, it generates a set of pairs $< T_i : m_i >$, where $\bigcup_{i=1} T_i$ denotes the type that describes the dataset and m_i counts the number of objects of type T_i. In the second phase, a type fusion

algorithm is applied to collapse similar types, which is based on some heuristics that depends of the value m_i.

3.2 NoSQL Schema Representations

In the past years, several metamodels have been proposed to represent NoSQL schemas. As indicated above, SOS [11] is a meta-layer (i.e. a metamodel) aimed to to support the heterogeneity of NoSQL systems. As shown in Figure 3.11, SOS provides a uniform representation for schemas of aggregate-based databases. A schema consists of a set of collections (*Set* metaclass). A key-value property is a simple element (*Attribute* metaclass) and a group of key-value pairs form a complex element (*Struct* metaclass). A Set can contain structs and attributes. Struct and Set can be nested.

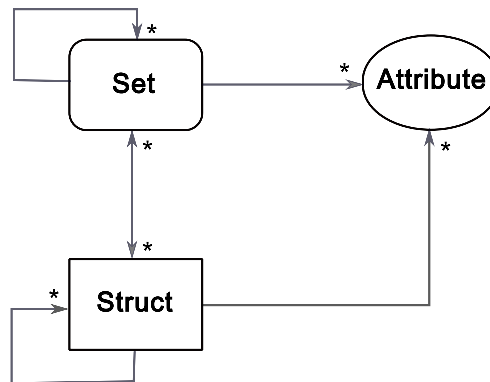

Figure 3.11: The SOS metalayer (extracted from [11]).

A design method for aggregate-based NoSQL database is proposed in [19]. This method defines the NoAM (*NoSQL Abstract Model*) model to represent these databases in a system-independent way. NoAM is really based on SOS and it has been designed to serve as an intermediate representation between aggregate objects of database applications and NoSQL systems. A NoAM database is a set of *collections* that contain a set of *blocks*. Each block is uniquely identified by a *block key*. Each block contains a set of entries that are key-value pairs. Several strategies to represent a dataset of aggregated objects as a NoAM database are proposed in [19].

Our *NoSQL_Schema* metamodel provides a higher level of abstraction than SOS and NoAM. These data models do not consider the possible existence of database object ver-

sions. Moreover, they does not explicitly represent relationships between entities. Moreover, they have not been implemented in form of a metamodel. It is worth noting that SOS and NoAM have been designed with a purpose different to our metamodel. Whereas we are interested in the schema discovering, SOS was devised for uniform accessing and NoAM is part of a design methodology.

As explained in Section 2.3, the *JSON Schema* format [4] has been recently proposed with the aim of providing standard specifications for describing JSON schemas. JSON Schema has been used to represent the schemas inferred in the approach of Klettke et al. [72] which has been described in this chapter. Our metamodel is more expressive than this standard format, since it considers aggregate and reference relationships and entity versions.

3.3 NoSQL Database Utilities

The immaturity of current NoSQL tools has been noted in the report [1] that we have commented in Chapter 1. Actually, there is a lack of tools for NoSQL systems capable of providing functionality similar to that available for relational systems. As we have indicated, the Dataversity's report identified three main categories of capabilities that should be supported in the NoSQL space tooling: diagramming, code generation, and metadata management. In Chapters 5, 6 and 7, we will present some database utilities that have been developed in this thesis for each of these three categories of functionality. Here, we describe some of the few NoSQL tools currently available for data modeling.

ER/Studio Data Architect ER/Studio Data Architect [47] has been the first commercial tool supporting some kind of NoSQL data model. Since mid-2015, this tool support a reverse engineering process for MongoDB databases, and schemas discovered are visualized as E/R-like diagrams. The schema discovery process is applied on a single collection, and the diagram obtained shows the union entity schema but references are ignored, that is, the diagram visualizes the root entity and the direct or indirectly embedded entities. Figure 3.12 shows the diagram obtained for the collection of movies of our running example.

Note that the reference relationship to *Director* is not shown. Collections are represented by rectangles, and nested objects by rectangles with rounded corners. This schema discovering process does not take into account the possible existence of entity versions.

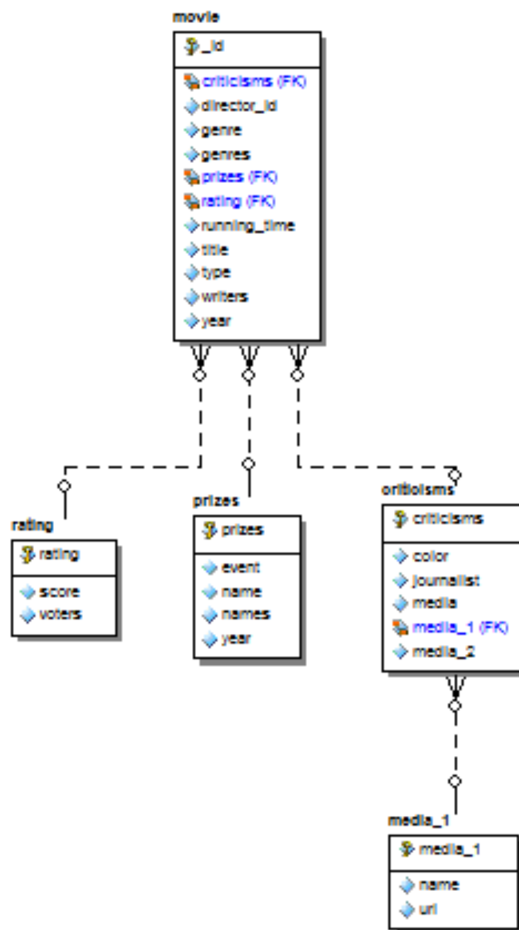

Figure 3.12: ER/Studio: Diagram for *Movie* collection.

CA ERwin CA ERwin* is another widespread modeling tool. ERwin Unified Data Modeler is a project under development with the aim of supporting data modeling for relational and NoSQL systems (Document and Column Family), which has been presented as an article in infoQ [48]. This tool will provide data schema discovery and data migration between RDBMS & NoSQL databases. A data model based on the Entity-Relationship notation is used to represent relational and NoSQL (Document/Column Family) logical schemas in a unified way. Entities, properties, relationships and keys are the main elements of such schemas. Entities represent tables and collections/column family; properties represent

*http://erwin.com/products/data-modeler.

columns and key/columns; relationships represent relational constraints and different elements in NoSQL schemas as References, Embedded Objects, and row across multiple column families; and keys represent indexes. A logical model also includes query patterns and data production patterns. Moreover, physical models for the three considered paradigms have been defined. The tool will be able to support forward and reverse engineering. Physical models can be automatically generated from a logical model, and a logical model can be automatically extracted and visualized from data. Query and data production patterns are used to transform logical models into physical models. With regard to the reverse engineering process to discover NoSQL schemas, the article [48] indicates that some techniques applied are: Schema coverage on statistics of records, machine learning, feedback collection from UI, dimension building in classifier, and continual improvement of accuracy in schema inference process. However, the author of this article does not give any details on the implementation of this process and how these techniques are really applied.

DBSchema DBSchema [37] is a modeling tool that allows to represent relational databases graphically. It offers, among other things, an interactive schema tool, a reverse engineering tool for databases, a random data generator once the schema is known, and a visual query builder. Recently, the tool has been augmented with support for MongoDB. The schema discovering is not able to perform inter-collection references automatically, but they can be created later using the utility. As some other utilities, it combines all the attributes of all the entity versions, and it is not clear how will it deal with conflicting properties in different objects. Figure 3.13 shows the result of inferring our example database.

MongoDB Compass MongoDB Compass[33] is a graphical tool that is included in MongoDB to facilitate the data exploration and manipulation. Compass offers a graphical user interface that allows the user to analyze the stored documents and visualizes information on the schema of the collections, as the frequency, types and ranges of fields, e.g. a histogram is used to show the frequency of values of a integer field. This tool provides other functionality that is not related to understand the implicit schema, such as (i) to graphically show information on the query performance; (ii) to visually work with geospatial data; (iii) a visual editor that makes it easier to perform CRUD operations; (iv) to write validation rules for data; (v) to visually build queries; and (vi) to offer information that helps to manage

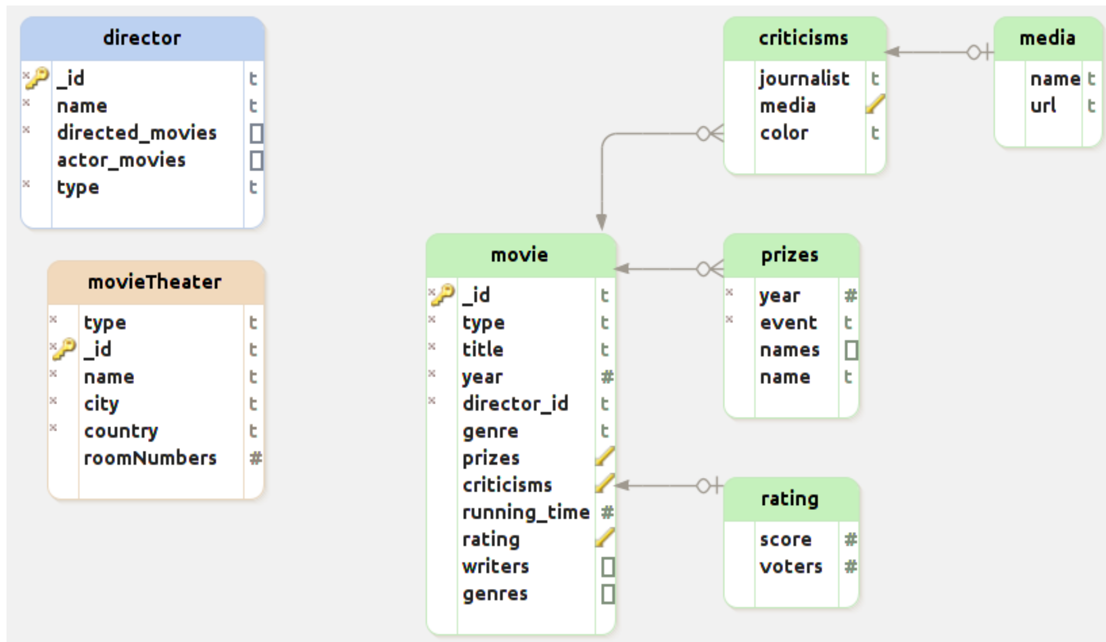

Figure 3.13: DBSchema Result for the Running Example Database.

indexes. As commented in discussing the work of Klettke et al. [72], we could also obtain statistics and outliers in our schema inference process.

Other Works An MDE approach to convert UML class diagrams to graphs inside graphs databases is presented in [36]. The 1authors build an intermediate GraphDB metamodel that is filled from the UML model using a M2M transformation. Then, Java artifacts that are able to manipulate the instances stored in the database are created. For all the classes in the UML model, CRUD classes are created to be able to create and manipulate entities of the UML classes stored in the graph database. An important point is that they are able to transform OCL restrictions present in the UML model into Gremlin code (a generic graph manipulation language supported by several graph based databases) to ensure the consistency of the graph. In Chapters 6 and 7 we will show how the schemas inferred in our approach can be used to automatically generate code for NoSQL database applications.

3.4 Final Discussion

In this section we shall contrast the schema inference process proposed in this thesis with the more relevant related works that have been discussed in this chapter. Table 3.1 summarizes the comparison attending to a set of dimensions established that appear in the leftmost column.

The notion of NoSQL schema presented in our work is more expressive than the JSON schemas in the standard. Our NoSQL schemas contain aggregation and reference relationships between entities, and also entity versions are extracted and represented. Our work therefore differs from the approach of Klettke et al. [72] in several essential aspects: i) the algorithm proposed identifies the required and optional properties, but object version schemas are not obtained; ii) schemas involve an *Object* type, but reference and aggregation relationships are not considered; iii) they do not specify how to cope with huge amounts of data; and iv) we obtain a model that conforms to a metamodel, instead of a JSON Schema.

The approach presented in this thesis differs of the work of Wang et al. [119] in the following aspects. First, the input of our schema extraction process is not the set of all documents stored into the database, but an array that only contains one object schema for each entity version of the database. This array of object schemas is obtained by applying a map-reduce operation. As explained in Section 4.1, an object schema has the same structure as the described object but each primitive value is replaced by a string or number that denote its JSON type. This pre-processing stage significantly improves the efficiency of our approach with respect to the algorithms presented in [72] and [119], in which all the objects stored are managed. Noting that our map-reduce operation provides the minimum number of objects needed to discoverer all the schemas existing in the database. In Chapter 5 we will explain in detail our MapReduce operation. Secondly, the output of our schema extraction process is a model that conforms to an Ecore metamodel that represent NoSQL schemas, which will be described in Section 4.2. The use of models entails two important benefits as are (i) a representation at a high-level of abstraction, and (ii) taking advantage of MDE technology to perform tasks as validation, automatic code generation and tool integration. In Chapters 6 and 7 we will show how MDE technology has been used to develop some database utilities. Thirdly, our output model registers all the discovered schemas and they are used for visualization and code generation (e.g. schemas for ODM mappers), instead Wang et al. focused

on generating a skeleton for the presentation. It should be noted that these authors consider scenarios where a large number of schemas can be discovered, whereas our work is focused on business applications scenarios in which the number of different schemas for an entity is limited. In such applications, when a new schema version for an entity is defined during a evolution of the system, data for the previous versions are migrated to the new version. This is due to the greater difficulty to write code and queries on several schema versions. A discussion of MongoDB developers on this topic can be found in [99].

The schema inference process considered in MongoDB-Schema [108] is very limited with respect to our proposal. Versioned schema are not considered and the process only discovers and extracts the union object schema of a collection. Aggregation and reference relationships are not considered. The schema inferred is represented as a JSON document, whereas we obtains Ecore models.

While Spark SQL [123] obtains union object schemas, our approach discovers and represents the set of versioned schemas defined in Section 4.1. In the case of union object schemas, the conflicting types are solved by defining union types. Thus, the schema inference of Spark SQL has the same limitations of MongoDB-schema. Our versioned schemas are complete, and allow having a more fine grained control of the objects that enter to and are obtained from a database. Moreover, the reference and aggregation relations between entities are not made explicit in Spark SQL.

The JSON schema discovering work of Cánovas and Cabot [64, 65] is close to our approach but there are some significant differences between them as indicated below. Unlike the works previously analyzed, this inference process extracts entity domain models rather union object schemas, which requires discovering and extracting the involved aggregation relationships. However, the existence of data versions (i.e. versioned schemas) is not addressed, and the references between objects are not discovered. This can be easily checked by trying visualize the schema for our Movie database example in [40]. Another remarkable difference of the inference process has to do with the strategy applied to resolve conflicting types in building an entity as the union of the schemas of the different versions. We extracted the union of all the types identified, instead of generalizing to the String type. It is remarkable that our work focused on NoSQL databases, whereas JSON Discoverer is a solution intended to developers that manage Web APIs. Therefore, we had to take into ac-

count some specificities of databases, such as the very large number of documents that can exist in a collection. Thus, a JSON model injection is not feasible in our solution, instead we have used a map-reduce operation as explained in Chapter 5. Moreover, we have extracted models that are instances of a metamodel that represents NoSQL schemas, instead of generating domain models that are instances of the Ecore meta-metamodel. Finally, it is worth to note that the JSON-to-Ecore mapping defined in the work of Cánovas and Cabot is similar to the first strategy defined in this thesis for obtaining a visual representation of the global schema, which will be explained in Chapter 6. We have converted the inferred schema models into Ecore metamodels in order to take advantage of the metamodel editor provided in Eclipse/EMF. However, in our case, the mapping is more direct.

With respect to ExSchema [23], we could remark the following differences: (i) our metamodel provides a representation at higher level of abstraction than the ExSchema meta-layer, (ii) Versioned schemas are not addressed in ExSchema, (iii) Aggregation and reference relationships are not explicitly differentiated, (iv) we visualize schemas in a way that facilitates the understanding of the data structure. Actually, data and code reverse engineering are complementary approaches. Our work could be integrated with a solution based on code analysis. It is worth to note that the source code analysis implemented by ExSchema focused on insert and update operations of the supported APIs, and the authors have not presented new contributions on this tool.

Our work contrasts with the XML schema extraction approaches here commented in several aspects. We have used a metamodel to represent the schemas, which has allowed us to apply MDE techniques, and we keep the different versions instead of obtaining a single schema. Our schema is richer, taking into account aggregations and references. With regard to the algorithm presented in [31], our inference process also has a first phase that applies a map-reduce operation, whose main purpose is to achieve scalability. The first step of this phase performs the same job that the map-reduce in [31], however in this work no abstract data model is generated.

In Chapter 6 we will present the diagrams proposed in this thesis to represent versioned schemas. Our solution visualizes the versioned schemas defined in Section 4.1. Instead, in ER/Studio [47] and DBSchema [37] the inference process is applied on a single collection, and the diagram obtained only shows the union object schema. Moreover, these tools only

support MongoDB at this moment, while any aggregate-based NoSQL system could be supported in ours.

Regarding to the project outlined in [48], we highlight the following differences: (i) our metamodel to represent NoSQL schemas is more expressive than the *Unified Data Model* proposed; (ii) our inference process extracts the schemas of all the documents stored; and (iii) we have defined diagrams for different kinds of versioned schemas. The separation between logical and physical data models is a strength of the ERwin's proposal. We are also applying intelligent techniques, such as algorithms based on decision trees, to classify the objects into entity versions. Finally, noting that our approach generates schema models that conform to an Ecore metamodel, so that we can take advantage of MDE techniques in order to develop utilities around them.

	Klettke et al.	Wang et al.	MongoDB-Schema	JSON Discoverer	ExSchema	Colazzo et al.	ER/Studio	ErWin	DBSchema	Our approach
Purpose	Schema extraction for NoSQL stores	Schema extraction for NOSQL stores	Schema extraction for MongoDB	Schema extraction for Web APIs	Schema extraction for NoSQL stores	Infer a succinct type	Schema extraction for MongoDB	Schema extraction for NoSQL stores	DB management tool & Schema inference	Schema extraction for NoSQL stores
Algorithm	Based on [79] (extraction of DTDs)	Clustering algorithm based on eSiBu tree	n/a	JSON model injection and M2M	Static code analysis	MapReduce + type fusion (greedy algorithm)	n/a	n/a	Propietary	MapReduce + Tailored Inference
Input	JSON collection	JSON collection	JSON collection	JSON data describing Web APIs	Code of applications	JSON collection	JSON collection	Several formats	Database connection	JSON files & NoSQL DB connections
Kind of schemas inferred	Union object schema	Union object schema	Union object schema	Domain model	Root Object Schemas	Succinct type	Union entity schema but references	Logical and Physical schemas	Union object schema	Version schemas. Entity schemas. Union schema entity
Schema representation	SG graph	eSiBu tree data structure	n/a	Ecore meta-metamodel	EXSchema Meta-layer	Not defined	n/a	Unified modelset	Internal	NoSQL_Schema metamodel (Ecore)
Output	JSON Schema	Collection skeleton (summarized schema)	JSON document (including values and statistics)	Domain model (instance of Ecore)	Instance Diagram. Spring Roo scripts	JSON document	Entity diagram	Diagrams for Logical and Physical schemas	Integrated into the tool	NoSQL_Schema model
Relationships	Aggregation	Aggregation	Aggregation	Aggregation	Aggregation	Aggregation	Aggregation	Aggregation & Reference	Aggregation & Reference	Aggregation & Reference
NoSQL Systems supported	Document, Key-Value	Document	MongoDB	n/a	Document, Colum family, Graph	n/a	MongoDB	Document, Column family, Key-Value, R	Document, SQL	Document, (Column family and Graph in the works)
Scalable	No	No	No	n/a	Code Analysis	Map-reduce phase	Information not available	Information not available	Information not available	Map-reduce phase
Implementation	n/a	n/a	npm project [108]	Web app	n/a	n/a	Commercial tool	Project under development	Commercial tool	Eclipse-based tooling
Testing	MongoDB (2 real world datasets)	MongoDB (5 real world datasets)	n/a	n/a	Indvalid app (MongoDB and JPA)	n/a	n/a	n/a	n/a	Stackoverflow
Schema Visualization	No	No	No	UML Class Diagram, UML Object Diagram	Schema Tree in PDF file	No	Entity diagram	Diagrams for Logical and Physical schemas	SQL-Like diagrams with groups	Several diagrams for Versioned Schemas
Applications for schemas	Statitics, Outliers	No	No	No	No	No	Data Modeling tool	Data Modeling tool	Several, integrated into the Application	Code generation for ODM mappers, Validator generator, Data Classifier

Table 3.1: Summary and comparison of related work.

Order and simplification are the first steps toward mastery of a subject – the actual enemy is the unknown.

Thomas Mann

4 Schemas for NoSQL Databases

In this section, we shall define the types of schemas that we have identified for aggregate-oriented NoSQL databases. Starting from the concept of semi-structured object, we will first define the concept of object schema, and then will introduce the concepts of entity and entity version, and the notion of *Versioned Schemas*. Once we have defined the different kinds of schemas, we shall describe the metamodel proposed in this thesis for represent such schemas. To end this section, we shall outline the general architecture of the proposed approach to infer schemas.

4.1 Entities, Versions of entities and Versioned Schemas

As explained in Section 2.2, an aggregate-oriented NoSQL database stores a set of semi-structured objects. The *document* term is commonly used to refer to database objects in the case of document databases (e.g. MongoDB). Next, we will define the concepts introduced in Chapter 2 in a more formal way.

Object (Root and Embedded) A database object O is composed of one or more fields (a.k.a. *attributes* or *properties*) p_i: $O = \{p_0, p_1, \ldots, p_m\}$. Each field p_i is specified by a pair

$< n_i, v_i >$, where n_i and v_i denote the name and value of the field, respectively. The value of a field can be:

- An atomic value (a number, string, or boolean).
- Another object, i.e. an embedded object inside the object which the field belongs to.
- A reference to another object; this is usually a string or integer that matches the value of a field of that other object referenced. The exact representation of the reference depends on the implementation.
- An array of values, which can be homogeneous or heterogeneous.

Below we show a *Movie* object which is part of the *Movie* database introduced in Section 2.5 as a running example. This object will be used to illustrate the definitions presented here.

We will use the *Root Object* term to distinguish to those objects that are not embedded into any object. We will assume that the target object of a reference must be a root object.

```
{
    "title": "Truth",
    "year": 2015,
    "director_id": "345679",
    "genre": "Drama",
    "rating": {
        "score": 6.8,
        "voters": 12682
    },
    "criticisms": [
        {
            "journalist": "Jordi Costa",
            "media": {
                "name": "El Pais",
                "url": "http://www.elpais.com"
            },
            "color": "red"
        },
        {
            "journalist": "Lou Lumenick",
            "media": "New York Post",
            "color": "green"
        }
```

```
    ]
}
```

This *Movie* object is a root object which has (i) three fields with atomic values: *title*, *year*, and *genre*, (ii) one field with an object value (*rating*); (iii) one field with a reference value (*director_id*); and (iv) one field with an array of objects (*criticisms*).

Object tree An object O can be represented by a tree-like structure that is defined as follows.

- The root node has a child node by each field of the object O; this node is labeled with "root".
- The nodes for atomic values or reference values do not have children (they become leaf nodes), and they are labeled with the value.
- The nodes for embedded objects have a child node for each field of the object, and they are not labeled.
- The nodes for arrays have a child node for each element included in the array, and they are labeled with the string "[]".
- The edges going out of object-nodes are labeled with the names of the fields of the object.
- The edges going out of array-nodes are labeled with a natural number from 1 to the size of the array.

The number of levels or depth of an object tree is determined by the highest level of nesting of the embedded objects. Each field has associated a **field path** that results of traversing the edges from the root to the node that corresponds to that field. We will denote a field path as the ordered sequence of labels of the edges traversed (i.e., names of fields) starting with the "root", and the labels are separated by the "." (dot) symbol. When a the value of a field f is an array, we will use the notation $f[i]$ to denote the element i.

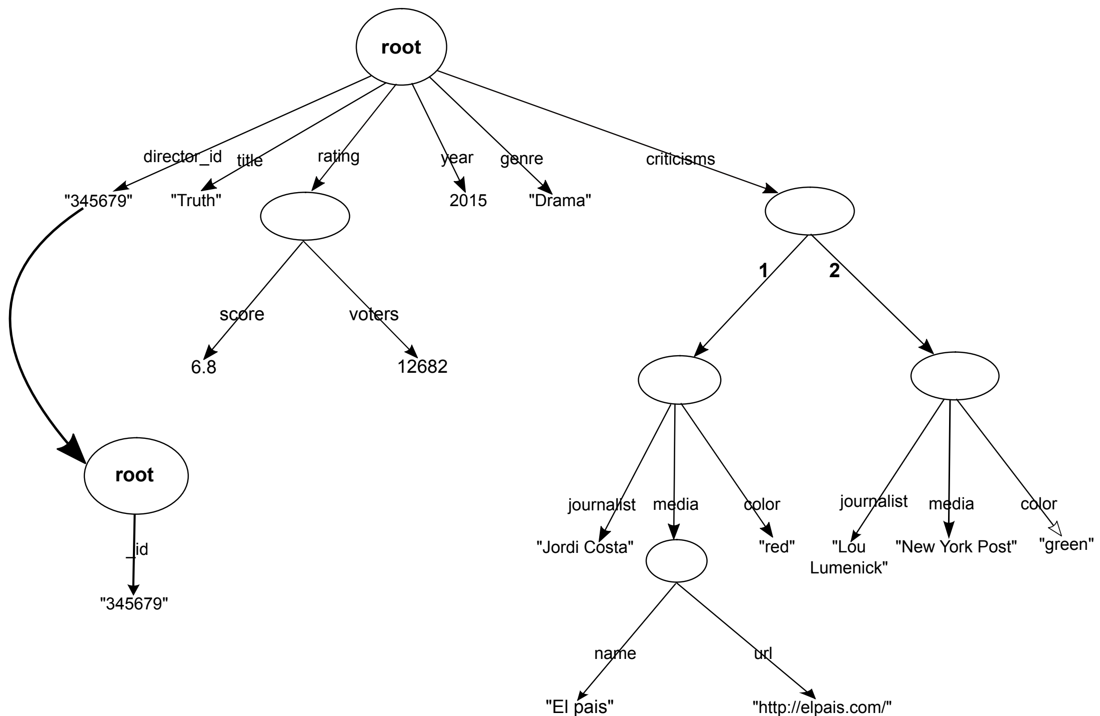

Figure 4.1: Tree for a Movie.

Figure 4.1 shows the tree for the Movie object shown above. This tree has four levels ($depth = 4$). Some examples of paths are *root.title* (level 1), *root.rating.voters* (level 2), *root.criticisms[1].journal* (level 3) and *root.criticisms[1].media.url* (level 4).

It is worth noting that the object tree is really a labeled directed graph due to the references among objects.

Object Schema The *schema* (or *type*) of an object is obtained by replacing the atomic values of the object by an identifier that denote its type (i.e. String, Number, or Boolean). Therefore, a schema has the same structure as the described object with respect to fields, nested objects and arrays. That is, a schema can also be represented by a tree-like data structure whose leaf nodes are labeled with type identifiers instead of atomic values, as shown in Figure 4.2. We refer to this structure as **type tree**. In NoSQL databases there are two kinds of objects: root and embedded. A schema can therefore describe both kinds of objects. We will also use the *raw schema* term to refer to object schemas.

Actually, the set of primitive types depends on the data representation language. We are

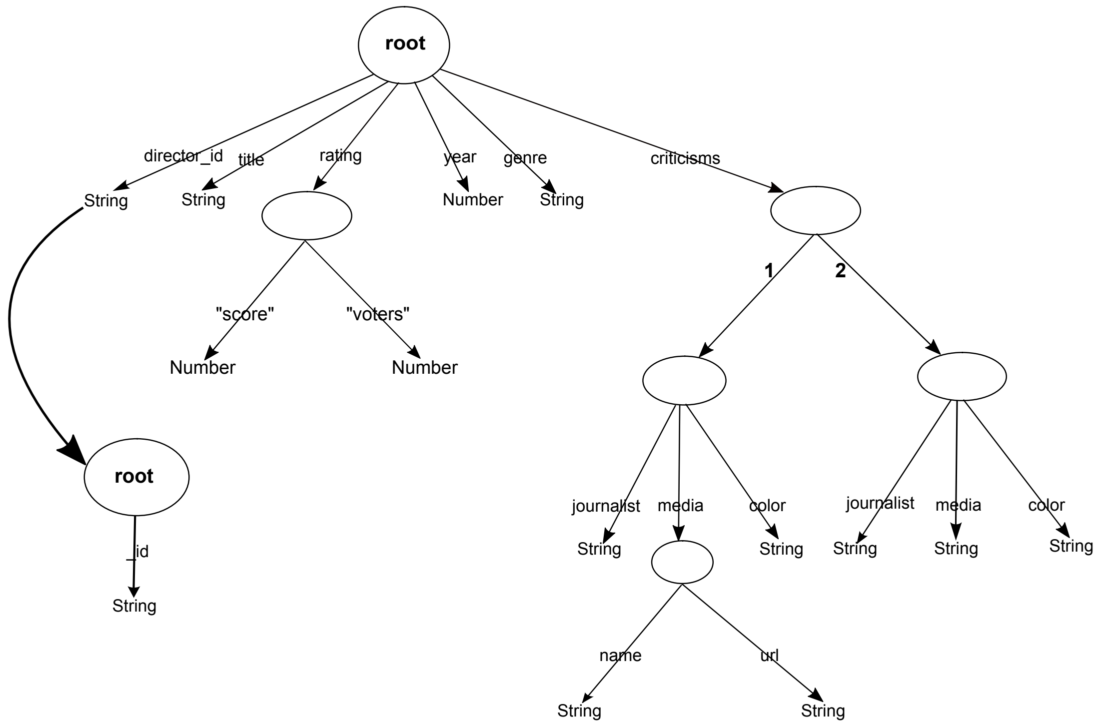

Figure 4.2: Type Tree for a Movie.

assuming that data are represented in JSON format and the primitives types are: *Number*, *String*, and *Boolean*.

The *Array* type will be the only kind of collection considered. The *tuple* term will be used to refer to arrays of atomic values (e.g. numbers or strings). We will use the notation $[T]$, where T denotes a data type, to express the type of a homogeneous array, and $[T_1, T_2, \ldots, T_p]$ to denote a heterogeneous array including p values v_i whose type is $T_i (i = 1, \ldots, p)$. Arrays can be nested. In Figure 4.2, the type of the *criticisms* field is an array of *Criticism* objects.

Entity and Entity Versions A database stores data that relate to entities of real the world (i.e any physical or conceptual thing that exists). Here, an *entity* labels all the objects that refer to the same concept (e.g. movie, director, or prize). As explained in detail in Section 2.6, the schemaless nature of NoSQL databases allows for different stored objects of the same entity type to have variations in their schemas. Therefore, we will introduce the notion of *entity version* to denote each of the sets of objects that, sharing the same entity label, have

a different schema. Each entity will have one or more entity versions. Note that versions exist both for root and nested entities. For example, in our *Movie* database example, there are 5 versions of *Movie*, 2 versions of *Director*, *Criticism*, *Prize*, and *MovieTheater*, and one version of and *Media*. *Movie*, *Director* and *MovieTheater* are root entities; *Criticism* and *Prize* are entities embedded into *Movie*; and *Media* is an entity embedded into *Criticism*. We refer to versions of an entity by means of the name of the entity followed of the version number that is preceded by an underscore symbol; for instance *Movie_4* and *Director_1* would be some of the entity versions in our database example.

Versioned Schemas The existence of *entity versions*, each with different levels of variation in their schema, raises the need for what we call *versioned schemas*, which denote the schemas defined for the set of entity versions. This is different from traditional databases or programming languages, where an entity has just one schema. Thus, each entity version will have its own schema, the *entity version schema*.

An *entity version schema* (or simply *version schema*) is obtained from the object schema of an entity version by replacing each embedded and referenced objects by the corresponding name of the target entity version or embedded object entity version, respectively.

Version schemas for root entities will be called *root schemas*. In Figure 4.3 we show (in JSON format) the root schema for the *Movie_4* entity version, to which the *Movie* object shown in Figure 4.1 belongs.

```
{
    "title": "String",
    "year": "Number",
    "director_id": "ref(Director_1)",
    "genre": "String",
    "ratings": "Rating_1",
    "criticisms": [
        "Criticism_1",
        "Criticism_2"
    ]
}
```

Figure 4.3: Schema for *Movie_4*.

A version schema therefore involves four kinds of entity types: (i) primitive; (ii) *reference*, when the field references objects of another entity; (iii) *aggregation*, when the field aggregates

objects of another entity; and (iv) *array type*, that can have one (homogeneous) or more (heterogeneous) base types that can in turn be primitive, aggregation, and reference types, and even another array type. The *relationship type* term will be used to refer both to the reference and aggregation types. Note that these two types originate that a schema is either directly or indirectly related to others. A version schema can also be represented in form of a tree-like structure as shown in Figure 4.4. In this tree root and nodes intermediate are labeled with the name of the corresponding entity version.

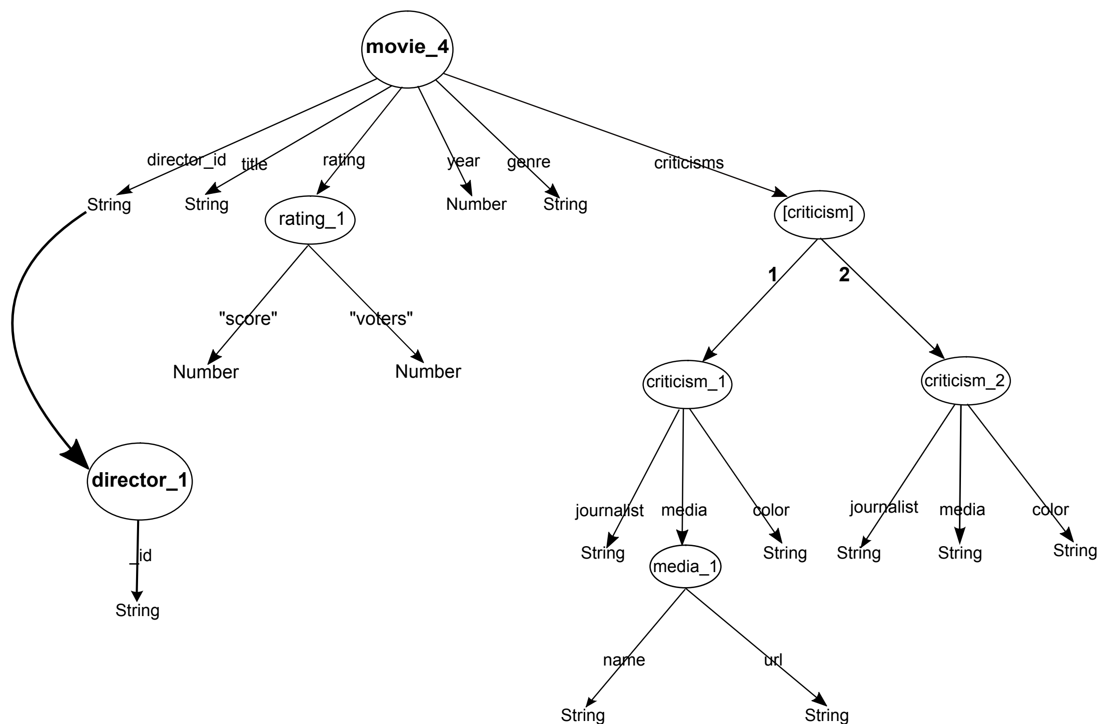

Figure 4.4: Version Tree for *Movie_4*.

In the *Movie_4* version schema, the *title*, *genre*, and *year* fields are of primitive type; the *director_id* field is of reference type; the *rating* field is of type *Rating*; and the *criticisms* field is of type array of *Criticism*. It is worth noting that the array type is not homogeneous, as it includes *Criticism* objects of two versions of the entity. As shown in Figure 4.4, the *Movie_4* version schema involves the *Criticism_1*, *Criticism_2*, *Director_1* and *Rating_1* version schemas, and these schemas involve in turn other schemas, in particular *Criticism_1* to *Media_1*), so that a schema graph is formed.

Next, all the version schemas that are direct or indirectly referenced by *Movie_4* are shown.

```
{
    "Criticism_1": {
        "journalist": "String",
        "media": "Media_1",
        "color": "String"
    }
}

{
    "Criticism_2": {
        "journalist": "String",
        "media": "String",
        "color": "String"
    }
}

{
    "Media_1": {
        "name": "String",
        "url": "String"
    }
}

{
    "directed_movies": [
        "Movie_4"
    ],
    "name": "String",
    "_id": "String"
}

{
    "Rating_1": {
        "score": "Number",
        "voters": "Number"
    }
}
```

Taking into account entity versions, an *entity schema* cannot be *just one* schema: it has to contain the set of schemas of the different entity versions of that particular entity. Thus, an *entity schema* can be defined as the set of schemas of their entity versions.

Sometimes, it is interesting to have a "view" of all the entity versions of a given entity. This can be done joining all the properties contained in the schemas of the entity versions of

the entity. Thus, an *entity union schema* can be constructed with the following rules:

1. For each property whose name appears only in one entity version schema, add that property to the *entity union schema*.

2. For each property whose name appears in more than one entity version schema:

   (a) If the type of the property is the same in all the entity version schemas in which it appears, add that property to the *entity union schema*.

   (b) If the type of the property differs in some entity version schemas, collect the set of different types of the property and build a *union type*. A union type of two types T_1 and T_2, denoted as $U(T_1, T_2)$, can be defined as a type that describes both the elements described by T_1 **and** those described by T_2.

Figure 4.5 shows the entity union schema for the *Movie* entity of our database in JSON format, and Figure 4.6 shows this entity schema in form of a tree.

```
{
    "title": "String",
    "year": "Number",
    "director_id": "ref(Director)",
    "genre": "String",
    "ratings": "Rating",
    "criticisms": [
        "Criticism"
    ],
    "prizes": [
        "Prize"
    ]
}
```

Figure 4.5: Entity Union Schema for the *Movie* Entity.

That is, an entity schema describes an entity by establishing its relationships with other entities, whereas a version schema describes a version entity by establishing its relationships with other entity versions.

It is worth noting that most of works on schema inference for JSON documents in NoSQL systems discover entity union schemas that result of superposing all the analyzed objects by replacing each primitive type value by its type. That is, the schema obtained is the union of

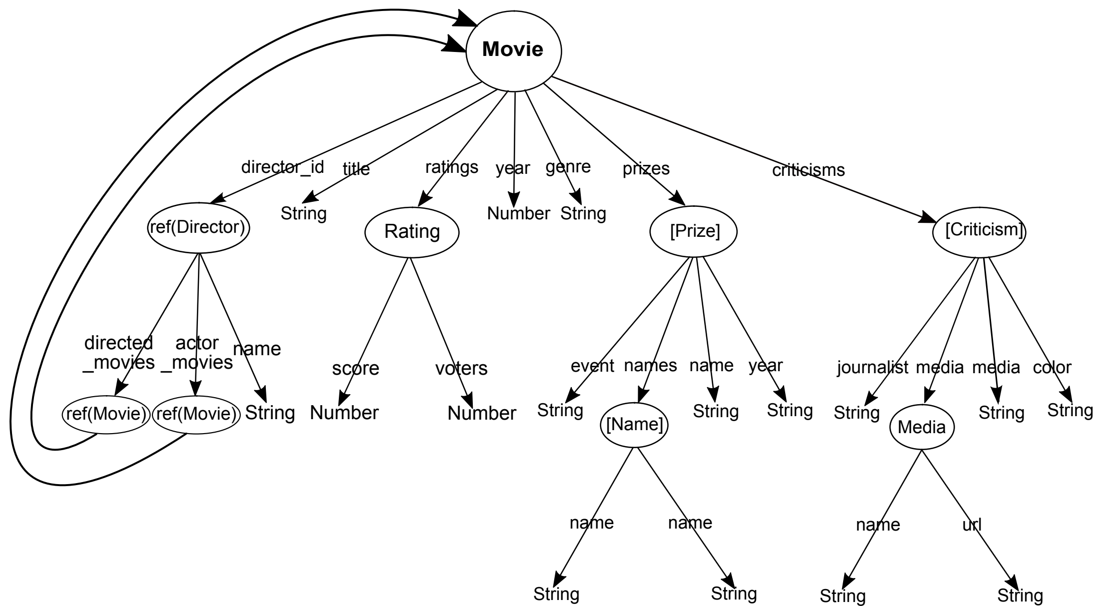

Figure 4.6: Tree of an entity union schema for Movie.

the object schemas of each entity version, but version schemas and entity schemas are not discovered and therefore the structure of the schema is similar to those shown in the tree in Figure 4.5. We refer to these schemas as *union object schemas*. Several strategies are used to solve the problem of conflicting types when a field belongs to more than one version. For instance, the type of the field can be the union of all the types encountered or the String type.

Finally we shall define two kinds of schemas for aggregate-oriented databases that involves all the entities instead of being defined for only one entity or entity version.

- **Database schema**. It is formed by the set of the *root schemas* that describe all the root entity versions of the database. As a schema recursively depends on the schemas of the embedded or referenced entities, a complete schema is formed by the set of version schemas of all the entity versions that exist in the database.

- **Entity database schema**. It is formed by the entity union schemas that describe all the root entities of the database.

4.2 A Metamodel for NoSQL Schemas

A key element of our proposal is the metamodel used to represent aggregate-oriented NoSQL database schemas. Figure 4.7 shows the metamodel that we have defined according to the definitions introduced in previous section. This metamodel will be called *NoSQL-Schema metamodel*.

A complete schema (metaclass *NoSQLSchema*) is formed by a collection of entities (*Entity*). Each entity has one or more versions (*EntityVersion*). A version has a set of properties (*Property*) that may be *Attribute*s or *Association*s, depending on whether the property represents a simple type or a relationship between two entities. A tuple denotes a collection that may contain atomic values and they can be nested. An association can be either an *Aggregation* or a *Reference*. The cardinality of an association is captured by the *lowerBound* and *upperBound* attributes, which can take values 0, 1, and −1. In addition to the set of properties (*properties* reference), an entity version has three attributes that are used to register whether it is a root entity (*root* boolean attribute), the number of documents in the database of that particular entity version (*count* long integer attribute), and the version identifier (*versionid* integer attribute).

Note that an aggregate is connected to one or more entity versions (*[1..*] refTo* reference) because an embedded object may aggregate an array with objects of different versions. Instead, a reference is connected to one entity (*[1..1] refTo*), since we need to know that a version holds references to a certain entity, but we decided not to cross object boundaries. The *opposite* self-reference in the *Reference* metaclass is used to make the relationship bidirectional, and specifies the other end.

4.3 Overview of the Architecture

Extracting versioned schemas (i.e. complete and root schemas) from aggregate-oriented NoSQL databases involves discovering entities, versions of each entity, and the fields and relationships (aggregations and references) of each version. Therefore, a reverse engineering algorithm for this task must traverse all the stored objects (i.e. root entities), and analyze their structure in order to harvest all the schema elements, that is, to discover the implicit schema into the data.

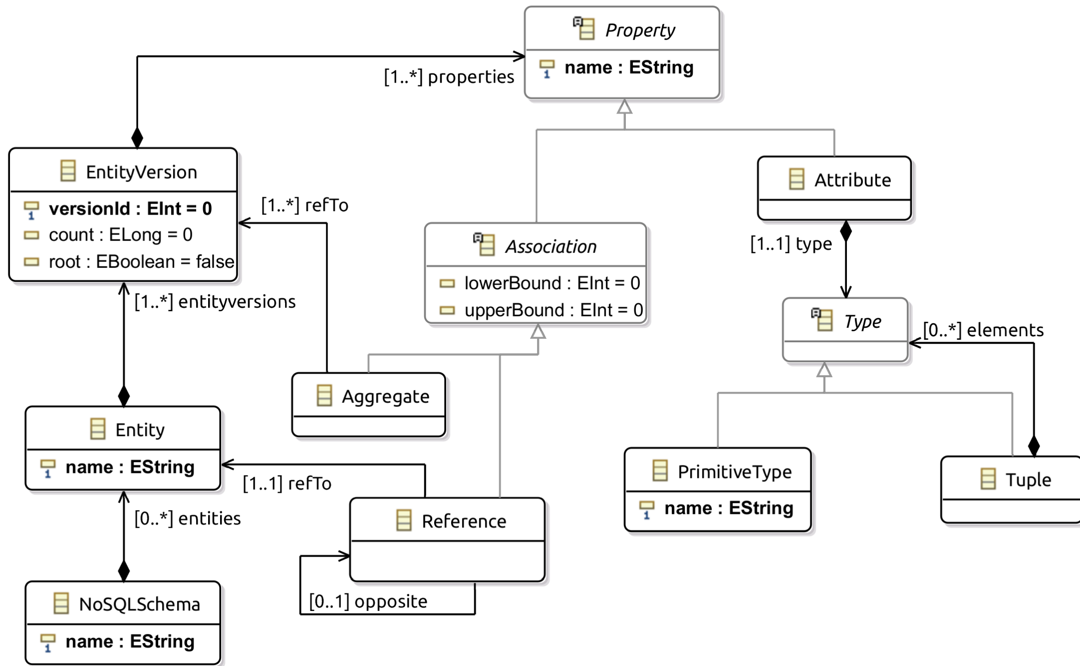

Figure 4.7: *NoSQL-Schema* Metamodel Representing NoSQL Schemas.

As explained in Section 2.9, reverse engineering can take advantage of MDE techniques: (i) metamodels provide a formalism to represent the knowledge harvested at a high-level of abstraction, and (ii) automation is facilitated by using model transformations. Therefore, we have devised an MDE solution to reverse engineer versioned schemas from aggregate-oriented NoSQL databases. As we will show in Chapters 6 and 7, these models inferred will be used to create database utilities.

Figure 4.8 shows the architecture of the MDE-based reverse engineering solution proposed in this thesis to extract versioned schemas, which is organized in three stages. Firstly, a MapReduce operation is applied in order to extract a collection that contains *one raw schema for each version of an entity* as explained in next section. This collection is named *Raw Schema Collection*. As indicated in Section 2.8, MapReduce is germane to most NoSQL databases, and provides a good performance as it is the native processing method when an algorithm has to deal with all the objects in a database. Thanks to the MapReduce operation, the reverse engineering process only must manage a small collection of simple objects instead of all the stored objects. This means a considerable optimization.

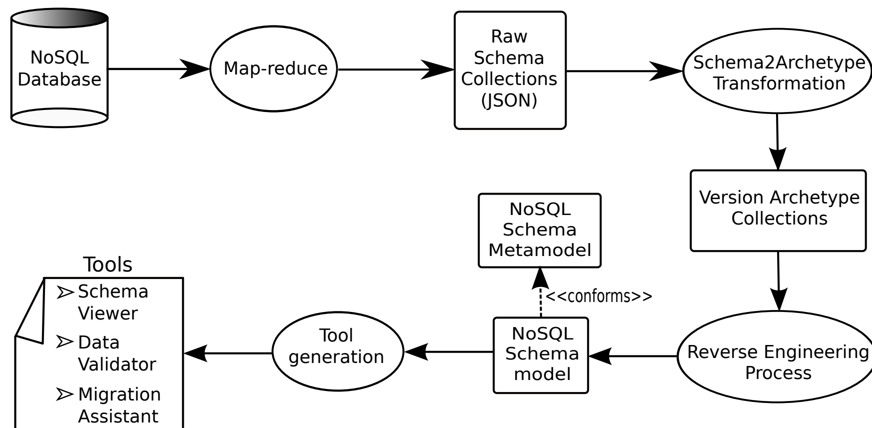

Figure 4.8: Overview of the Proposed MDE Architecture.

Secondly, the raw schemas of the collection obtained are transformed into a JSON representation. We will use the *Version Archetype Collection* term to refer to this new collection.

Thirdly, the reverse engineering process analyzes all the JSON objects of the Version Archetype Collection and generates a model that conforms to the NoSQL-Schema metamodel (Figure 4.7).

As shown in Figure 4.8, the inferred NoSQL-Schema models may be used to build tools that could be classified in two categories: i) database utilities that require knowledge of the database structure, for instance a SQL query engine for NoSQL databases or database statistics, and ii) helping developers to deal with problems caused by the absence of an explicit schema, for instance the tools presented in Chapters 6 and 7, which can generate data validators or schema diagrams, among other applications. M2M and M2T transformations have been used to implement these tools.

Have a vision. It is the ability to see the invisible. If you can see the invisible, you can achieve the impossible.

Shiv Khera

5 The Inference Process of NoSQL_Schema Models

This chapter describes the inference process defined in this thesis to extract implicit versioned schemas of aggregate-oriented NoSQL databases. Our strategy takes into account the existence of *versions* of the entities as well as *references* among entities. The inference process is organized in three stages as commented in Section 4.3. We shall explain each stage in detail and show the validation of the inference process that has been performed.

5.1 Obtaining the Version Archetype Collection

To improve the efficiency, we have considered a preliminary stage that applies a *MapReduce* operation to obtain a collection that only contains one raw schema for each entity version of the database. This collection will be referred to as the *Raw Schema Collection*.

For each database object (i.e. root objects), the *map()* operation performs a two-step process. First, it generates the *version identifier*. The *version identifier* is obtained by changing the values from all the key/value pairs of the object with their corresponding type, so a pair "`"name":"Pedro"`" is converted to "`"name":"s"`", the "s" indicating that that particu-

lar value was a String. The special *type* field is left intact. To this object, an object-to-string serialization is performed, that becomes the *version identifier*. Secondly, the <*version identifier, {count:1}*> key/value pair is emitted. The value is an object used as accumulator of the number of objects of a particular entity version.

Recall that a *raw root schema* of a root object was defined in Section 4.1 as a JSON object built honoring two rules: i) it has the same structure as the root object with respect to fields, nested objects and arrays, and ii) each primitive value in the root object is replaced by its JSON type.

In our running example (Figure 2.8), *{name:String, directed_movies:[String],actor_movies:[String]}* would be the raw schema for the *Director* entity with *_id=123451*, and *{title:String, year: Number, director_id:String, genre: String, criticisms:[{journalist:String, media:String, url: String, color: String}]}* would be the raw schema of the *Movie* with *_id=4*. More visually:

JSON object	**Raw Schema**
{name: "Orson Welles",	*{name:String,*
directed_movies: ["1", "5"],	*directed_movies:[String],*
actor_movies: ["1","5"]	*actor_movies:[String]*
}	*}*

Note that the *type* and *_id* fields have not been considered for sake of simplicity.

Next, we show the key/value pair generated by the *map()* operation for the *Movie* object with *_id=4*. The key (i.e. the version identifier) is formed by *Movie* followed by the raw schema of the object, and the value is the *Movie* object. One object with this structure would be generated for each object stored into the database.

```
{"{type:"movie",title:"s",year:0,director_id:"s",
genre:"s",criticisms:[{journalist:"s",media:"s",
url:"s",color:"s"}]}":{count:1}}
```

Note that "`"s"`" is used for Strings, "`0`" is used for numeric values, and "`true`" for boolean.

Once the *map()* generates the key/value pairs with the above indicated structure, the *reduce()* operation is performed once for each version identifier. It receives a key (version identifier) and a set of counters, and calculates the total number of objects. The result is an array of serialized JSON objects, but now containing just one object per version entity and the *count* field accumulates the total number of objects for the corresponding version.

The *MapReduce* operation is followed by a transformation that converts strings representing serialized JSON objects into *archetype* objects with the structure of the corresponding entity version extracted from the original JSON objects.Therefore the Raw Schema Collection is converted into the *Version Archetype Collection*, which contains one archetype object per entity version.

5.2 Obtaining the Schema

The objects of the *Version Archetype Collection* are analyzed to discover the elements of the *complete schema* of the database: entities, versions, fields, and relationships. This analysis involves a reverse engineering process that must obtain a root version schema for each existing entity version. The task to be performed consists on transforming a labeled tree into a labeled graph. The input is a labeled tree that represents an archetype object of a entity version. This kind of structure was described in Section 4.1 and an example is shown in Figure 4.1. The target graph would be formed by (i) a tree of identical structure but intermediate nodes are labeled with a symbol denoting the inferred type, and (ii) the references from intermediate nodes to root nodes of another trees are established (i.e. object references are discovered). Figure 5.1 illustrates this transformation by showing how the tree for the Movie with *_id=4* (actually the archetype object is analyzed) is converted in the graph that represents the root version schema for this object.

Therefore, the reverse engineering process discovers the elements of a complete schema by analyzing the archetype object collection. The result is a model that conforms to the NoSQL-Schema metamodel introduced in Section 4.2.

As the schema elements are discovered, they are represented as elements of the NoSQL_Schema model generated as output. For this, the algorithm is organized in two stages. First, *Entities*, *EntityVersion*s, *Attribute*s, and *Type*s (*PrimitiveType*s and *Tuple*s) are created, and then *Association*s (*Aggregate*s and *Reference*s) are created in a second stage, once all the *EntityVersion*s have been discovered. Next, we shall describe in detail how this algorithm works.

Discovering entities and entity version for root objects The algorithm traverses all the objects of the collection. The value of the *type* field of archetype objects is used to create the *Entity* elements. An *Entity* will be created when a new value of the *type* field is found in the

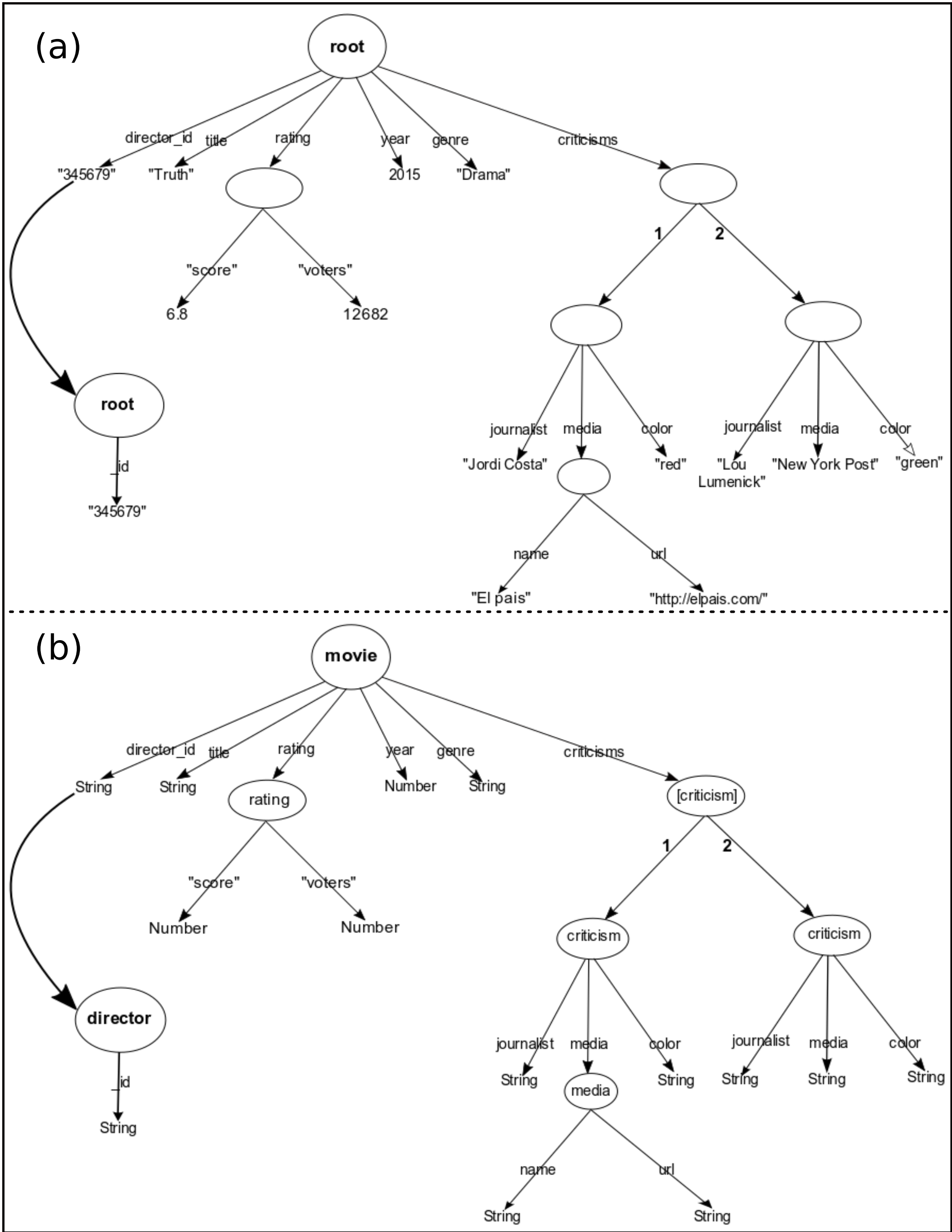

Figure 5.1: Schema Tree for Movie object with *_id* = 4.

visited object. Its name will be the value of the *type* field.

Each object in the archetype object collection should originate an *EntityVersion*. However, this does not always occur, because an *EntityVersion* may exist already that only differs in the cardinalities of one or more fields. In these cases, the cardinalities of the existing *EntityVersion* are adjusted to include both specifications, and no new *EntityVersion* is created.

Each *Entity* holds a list of entity versions, in which each new *EntityVersion* is added.

The treatment applied to each object of the collection involves an analysis of each of its pairs, more specifically the data type of the value is checked to discover model elements.

Discovering entity versions for embedded objects When the value of a pair is an object or a tuple of objects, then such objects are recursively traversed to discover their entity version types. Such types are identified by obtaining its raw schema and the name of the corresponding entity. This name is obtained from the pair's key. If the value is an array of objects and the name is plural, then the singular name is used. We keep a collection of existing raw schemas for each discovered entity. Then, we check if the new raw schema discovered already exists in the collection of the corresponding entity. If it does not exist, a new *EntityVersion* is created and added to the associated *Entity*. When the created *EntityVersion* is the first one discovered for a particular entity, an *Entity* element is also created. Several *EntityVersion*s may embed the same aggregated *Entity*.

An *EntityVersion* is named by appending, to the entity name, a suffix with an underscore and a counter of the number of version. For instance, two *EntityVersion* would be generated for the *Director* root objects of the running example, named *Director_1* and *Director_2*, and *Prize_1* and *Prize_2* would be generated for the *Prize* objects embedded into *Movies*. Figure 5.2 shows a textual report with all the entity versions generated for the running example.

Discovering Attributes and Types An *Attribute* is generated for each visited object's pair whose value is either atomic or an array of either primitive types or nested arrays of primitive types. The attribute name is given by the pair name. With regard to the type, a *PrimitiveType* or a *Tuple* is generated depending on whether the value is atomic or an array. Each created *Attribute* is added to the collection of attributes of the corresponding *EntityVersion*. For instance, the pair *"title":"Truth"* in a version of *Movie* would lead to the *Attribute* named "title" and a *PrimitiveType* named "String"; and a pair *"nationality": ["Spanish",*

```
Versions Entities:
Entity Movie {
 Version 1 {
  genre: String
  title: String
  year: int
  director[1]: [Ref]->[Director] (opposite=true)
  criticisms[+]: [Aggregate]Criticism
  prizes[+]: [Aggregate]Prize
 }
 Version 2 {
  genre: String
  title: String
  year: int
  running_time: int
  director[1]: [Ref]->[Director] (opposite=true)
 }
 Version 3 {
  genre: String
  title: String
  year: int
  director[1]: [Ref]->[Director] (opposite=true)
  prizes[+]: [Aggregate] Prize
 }
 Version 4 {
  genre: String
  title: String
  year: int
  rating[1]: [Aggregate] Rating
  director[1]: [Ref]->[Director] (opposite=true)
  criticisms[+]: [Aggregate] Criticism
 }
 Version 5 {
  genres: Tuple [String]
  title: String
  writers: Tuple [String]
  year: int
  director[1]: [Ref]->[Director] (opposite=true)
 }
}

Entity Movietheater {
 Version 1 {
  city: String
  country: String
  name: String
 }
 Version 2 {
  city: String
  country: String
  name: String
  roomNumbers: int
 }
}

Entity Media {
 Version 1 {
  name: String
  url: String
 }
}

Entity Rating {
 Version 1 {
  score: int
   voters: int
 }
}

Entity Director {
 Version 1 {
  actor_movies[+]: [Ref]->[Movie] (opposite=true)
  directed_movies[+]: [Ref]->[Movie] (opposite=true)
  name: String
 }
 Version 2 {
  directed_movies[+]: [Ref]->[Movie] (opposite=true)
  name: String
 }
}

Entity Criticism {
 Version 1 {
  color: String
  journalist: String
  media[1]: [Aggregate] Media
 }
 Version 2 {
  color: String
  journalist: String
  media: String
 }
}

Entity Prize {
 Version 1 {
  event: String
  names: Tuple [String]
  year: int
 }
 Version 2 {
  event: String
  name: String
  year: int
 }
}
```

Figure 5.2: Textual report of all the entity versions.

"French"] in a version of *Movie* would generate an *Attribute* named "nationality" and a *Tuple*.

Discovering Aggregation relationships A pair results in an *Aggregate* (i.e. an aggregation relationship) if its value is either an object or an array of objects. That is, an *Aggregate* is created for each *EntityVersion* that corresponds to an embedded object. Each created *Aggregate* element must be added to the collection (*properties*) of the corresponding *EntityVersion*, and must be connected to the *EntityVersion* (*refTo* reference).

Regarding to the cardinality, the *lowerBound* and *upperBound* attributes of *Aggregate* take their values depending on the multiplicity of the *Pair*, e.g. it is *one-to-one* (*lowerBound=1* and *upperBound=1*) if the pair value is an object that can not be *null*, and the cardinality is *zero-to-many* (*lowerBound=0* and *upperBound=−1*) if the pair value is an array of objects that can take the *null* value.

Discovering Reference relationships As explained in Section 2.3, a reference implies that an entity's pair identifies an object of another entity. That is, the pair values of the referencing entity match the values of another pair in the referenced entity (this is equivalent to foreign keys and joins in relational tables). These identifier values can be strings, integer numbers or arrays of these two primitive types. Two strategies are applied to discover references (i.e. *Reference* elements):

- Some conventions commonly used to express references are checked, such as:
  - If a pair name has the *entityName_id* suffix, then, a entity named *entityName* would be referenced if it exists.
  - MongoDB itself suggests to use a construct like *{$ref: "entityName", $id: "reference_id"}* to express references to objects of the entity named "entityName" [98].
- If a pair name is the name of an existing entity and the pair values match the values of a *_id* pair of such an entity.

For instance, the *directed_movies* field of *Director* entity versions references to an array of *Movie* objects, and the *director_id* field of a *Movie* entity versions references a *Director* object (Figure 5.3).

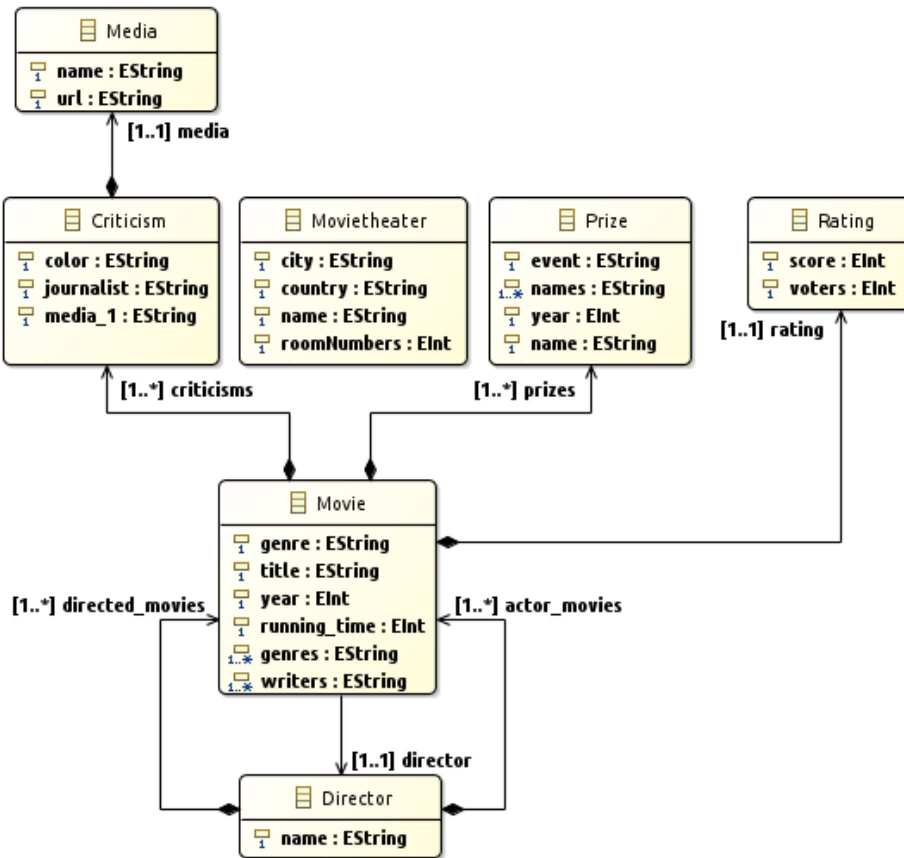

Figure 5.3: Graphical Representation of all the Entities with the sum of all fields.

As in the case of aggregations, the references are connected to the corresponding entity (*properties* and *refTo* relationships) in the second stage of the transformation. The cardinality is obtained for references as explained above for the aggregation relationships. Once all the references have been generated, the *opposite* relationship is resolved.

5.3 Inference Validation

We have done some tests to validate the inference approach. To test the inference algorithms, we used several datasets, as well as some synthetic datasets generated with a random data generator that reproduces the structure of a given NoSQL Schema.

5.3.1 Roundtrip Validation

For the validation of the inference algorithm and the implementation, we generated several artificial test cases to exercise all the different combinations of entities, versions, attribute sharing, changes in types, references, etc.

We developed a random data generator that, from a NoSQL_Schema model, generates a complete database, supporting both MongoDB and CouchDB. The inference process is then run against the database, and the same NoSQL_Schema model should be generated as output. The generated model can change the entity version numbers, but the same entity versions must be generated, with the same properties, aggregations and references.

The data generator itself is interesting to generate example databases given a desired schema, that can allow the programmers to better assess the database properties with respect to indexes and queries.

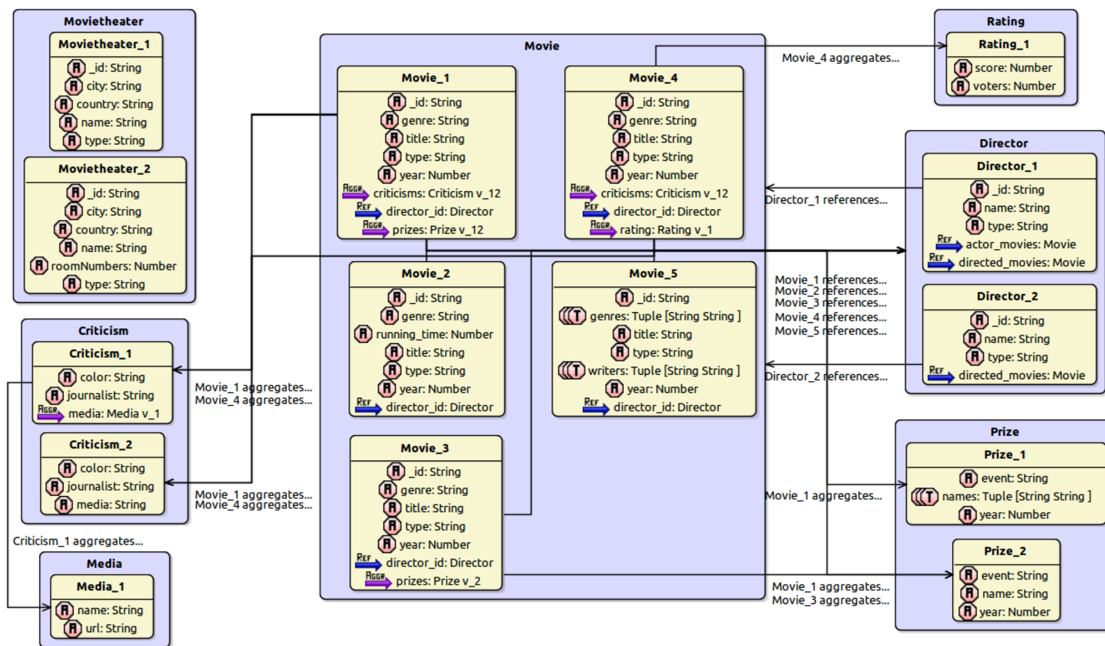

Figure 5.4: Global View of the Movie Versioned Schema (Original Model).

Figure 5.4 shows the original schema, while Figure 5.5 shows the inferred schema after the database population. Note how there are the same entities and versions, with the same properties, the only visible change being that *Movie_2* and *Movie_4* are exchanged.

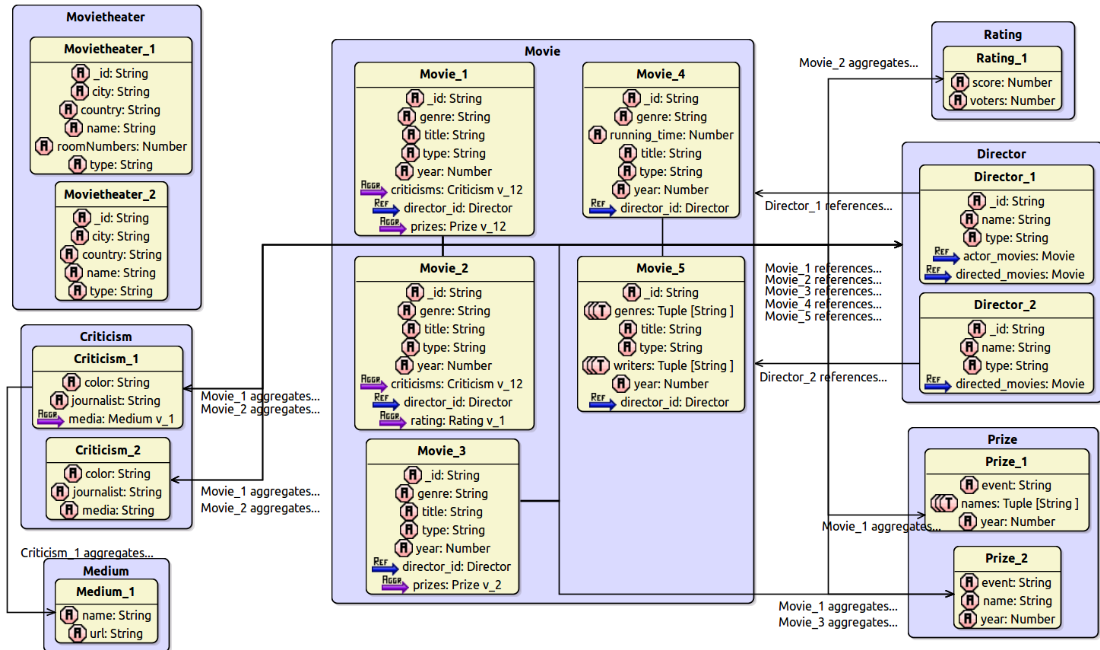

Figure 5.5: Global View of the Movie Versioned Schema (Inferred Model).

5.3.2 Scalability Case Study: StackOverflow Database

To show the validity of our approach with a real world database, we searched for public databases that had evolved during time, adding and removing attributes, having optional attributes, etc.

We found that the StackOverflow site[*] has public data dumps of all its content, including posts and answers, post history, users, comments, tags, badges, etc. This website has been working since 2008, and was founded by well-known software community gurus, Jeff Atwood and Joel Spolsky. With almost ten years of service, their database has evolved during time, and the format in which they offer the data dumps (XML)[†], allows to capture the differences in each of the elements of the different entities.

The dataset is medium-sized, and consists of several XML files, one per entity (users, posts, tags, etc.), which in total sum about 100 GB compressed. It contains, at the time of this writing (March, 2017) around ten million posts and three million users.

The process of data loading was done using MongoDB 3.4.2 in an Intel(R) Core(TM) 2

[*] http://stackoverflow.com/.

[†] https://archive.org/details/stackexchange.

↓ seconds objects →	0.5M	1M	2M	5M	10M	15M
Total insertion time	223	434	828	2865	4210	5554
Users	41	80	143	356	693	968
Votes	28	57	113	294	580	823
Comments	31	62	125	339	632	887
Posts	45	85	181	438	896	1188
Tags	17	18	18	17	17	17
Postlinks	30	69	125	285	613	835
Badges	31	63	123	308	623	836
MapReduce & inference	39	73	138	336	652	919
Total time	262	507	966	3201	4862	6473

Table 5.1: Insertion, MapReduce, and inference times for different dataset size.

Quad CPU Q8300 @ 2.50GHz, with 4 GB of memory. For data loading, a custom XML to MongoDB Java program was used. Each XML object was converted directly, selecting the different attributes and converting them into key/value pairs of JSON objects for MongoDB.

We tried different sizes to show the variation of insertion time and inference time. The table 5.1 shows the different times for different number of objects.

Figure 5.6 shows the graph representation of Table 5.1. In those times, it is interesting to note that the ratio between number of objects and time of the MapReduce and inference operations goes down as the number of objects increases, as shown in Figure 5.7. This is one of the indications that the algorithm for obtaining the schema is scalable, and again we only tried with one computer.

For the tests, we took the same number of objects of each entity type (except for Tags, that we always inserted them, as they only have 50,000 elements) to process from half million elements to fifteen million elements. The time for insertion and inference scales up linearly with the number of objects, and this was tested just in one computer. We expect to have speedup when more computers are used.

The inference process correctly determined the different entities in the original database. It also correctly determined the references between the different entities. However, some fields of Number type were incorrectly identified as references, for example, the *UpVotes* and *DownVotes* fields were identified as references to the *Votes* entity. We expect, as a future work,

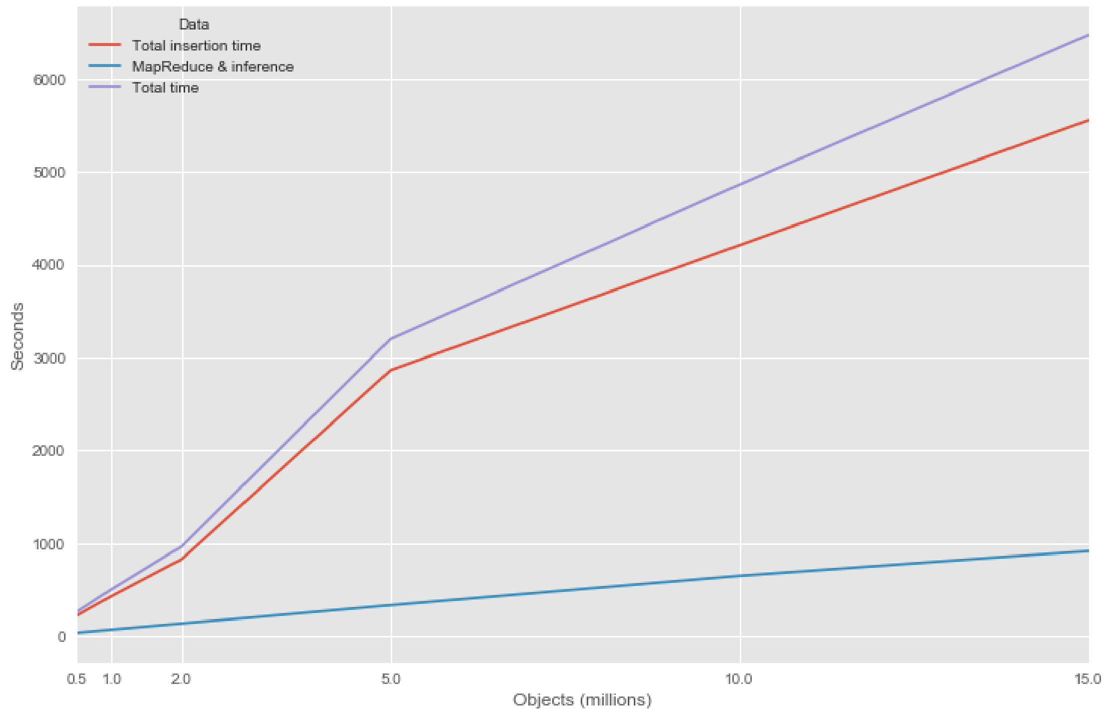

Figure 5.6: Graph of Times of Table 5.1.

to allow the user to specify which identified references are not references in fact. This dataset didn't contain aggregations, as it was obtained as a dump of a SQL database. Finally, it also detected the different versions for each entity.

Concretely, in the bigger 15 million objects example, 1 entity version was discovered for *Badges*, 4 versions for *Comments*, 263 entity versions for *Posts*, 4 for *Votes*, 33 for *Users*, etc. Figure 5.8 shows the generated model. The higher number of versions, that in many cases are due to optional attributes, gives us the idea of developing some capabilities for browsing and/or querying in the schema visualization tools. We leave it as a future work, in particular to extend the visualization tool created with Sirius that is commented in the Chapter 6.

Figure 5.9 shows the global set of entity versions discovered, using Sirius, while Figure 5.10 shows a union schema for the Stackoverflow example.

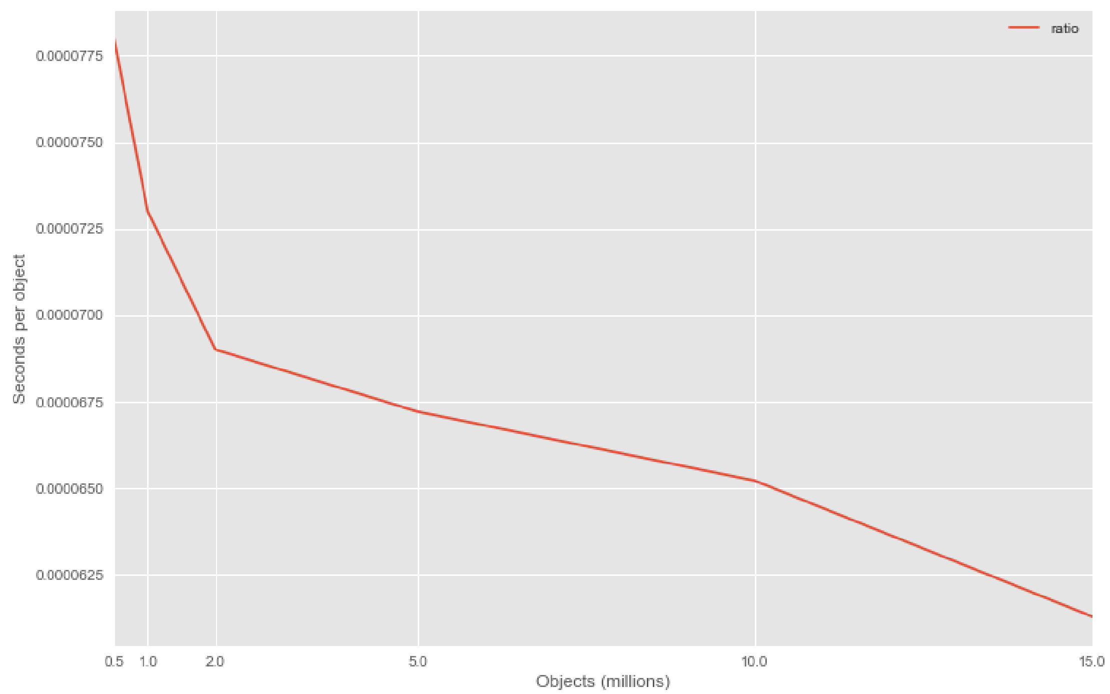

Figure 5.7: Ratio of MapReduce and inference time versus number of objects from Table 5.1.

- No SQL Schema stackoverflow
  - Entity Badges
    - Entity Version 1
  - Entity Comments
    - Entity Version 1
    - Entity Version 2
    - Entity Version 3
    - Entity Version 4
  - Entity Posts
  - Entity Votes
    - Entity Version 1
    - Entity Version 2
    - Entity Version 3
    - Entity Version 4
  - Entity Users
  - Entity Postlinks
    - Entity Version 1
  - Entity Tags
    - Entity Version 1
    - Entity Version 2

Figure 5.8: NoSQLSchema model result for Stackoverflow.

Figure 5.9: Global Set of Entity Versions Generated for Stackoverflow.

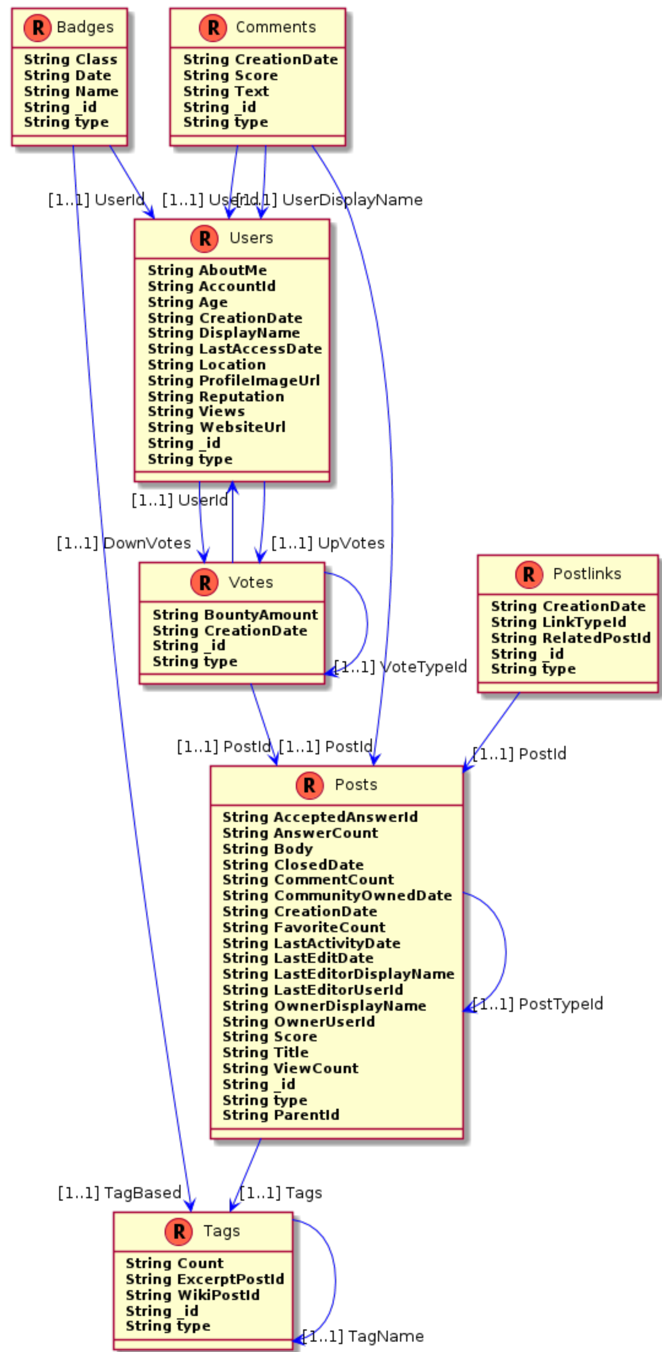

Figure 5.10: Union Schema using PlantUML for the Stackoverflow Example.

Visible things can be invisible. However, our powers of thought grasp both the visible and the invisible – and I make use of painting to render thoughts visible.

René Magritte

6 Visualization of NoSQL Schemas

In Chapter 5 we explained the approach devised in this thesis to infer schemas for NoSQL databases. In this Chapter and the next one we shall present some applications of the inferred schemas. In particular, we have developed solutions to visualize schemas and automatically generate different kinds of software artifacts for NoSQL database applications: code for ODM mappers, data validators, and object classification into entity versions. These database utilities mitigate problems that lack of explicit schemas cause to developers.

In this Chapter, we will describe three approaches to visualize NoSQL schema diagrams: (i) to convert Schema models into Ecore models (i.e. metamodels), and using the metamodel editor available for Ecore/EMF, which represent Ecore metamodels as UML class diagrams; (ii) to generate PlantUML code for visualizing UML class diagrams; and (iii) to define a specific notation. The two first proposals have been devised in this thesis, and the third one has been developed in a Master's Thesis based on the results obtained in our work [29].

6.1 Visualization of NoSQL Schemas

The activity of software modeling brings four important benefits: (i) models help to reason and understand the system under study; (ii) models facilitate the communication among stakeholders; (iii) models document the design choices; and (iv) models can be formally

specified in order to generate artifacts of the final application. These benefits are also gained with NoSQL schema models, in particular when they are visually represented. Schema diagrams are useful both to designers and developers. Designers can express the database structure at a high level of abstraction, and developers can write better code if they have a model that properly represents the database schema. Moreover, schema diagrams provide a very useful documentation that facilitates the database evolution.

NoSQL-Schema models inferred contains all the information needed to express the different versioned schemas defined in Section 4.1. We have developed several utilities to visualize diagrams that represent to each kind of versioned schema.

Initially, we used *EMF to Graphviz* (emf2gv) [45] to test our inference process. This tool is a Eclipse plugin that generates a diagram that represents the object graph of an EMF model. Figure 6.1 shows an excerpt of the diagram generated for our Movie database. This is a representation similar to those offered by some schema discovering tools as [23]. i.e. a schema tree. However, NoSQL schemas should be represented by means of a more adequate notation that clearly shows the schema elements: entities, entity versions and relationships. UML class diagrams can be used to represent NoSQL schemas just as they are used for relational schemas.

In this thesis, we have built visualization tools that show versioned schemas (root version schemas, entity union schemas and entity database schemas) as UML class diagrams. We first developed a simple solution to visualize entity database schemas by taking advantage of the metamodel editor included in Eclipse/EMF [102]. After, we used plantUML [96] to generate diagrams for representing root version schemas, union entity schemas and entity database schemas. This experience evidenced the convenience of defining some specific notation to adequately visualize NoSQL versioned schemas. Such a notation was implemented by means of the Sirius tool [7] as part of a Master's Thesis [29]. Next, we will describe each one the diagrams proposed for NoSQL schemas, as well as the solutions implemented for their generation.

6.2 Using EMF Metamodel Diagram to Visualize Entity Union Schemas

Figure 6.2 shows the first MDE solution that we have devised to visualize entity union diagrams.

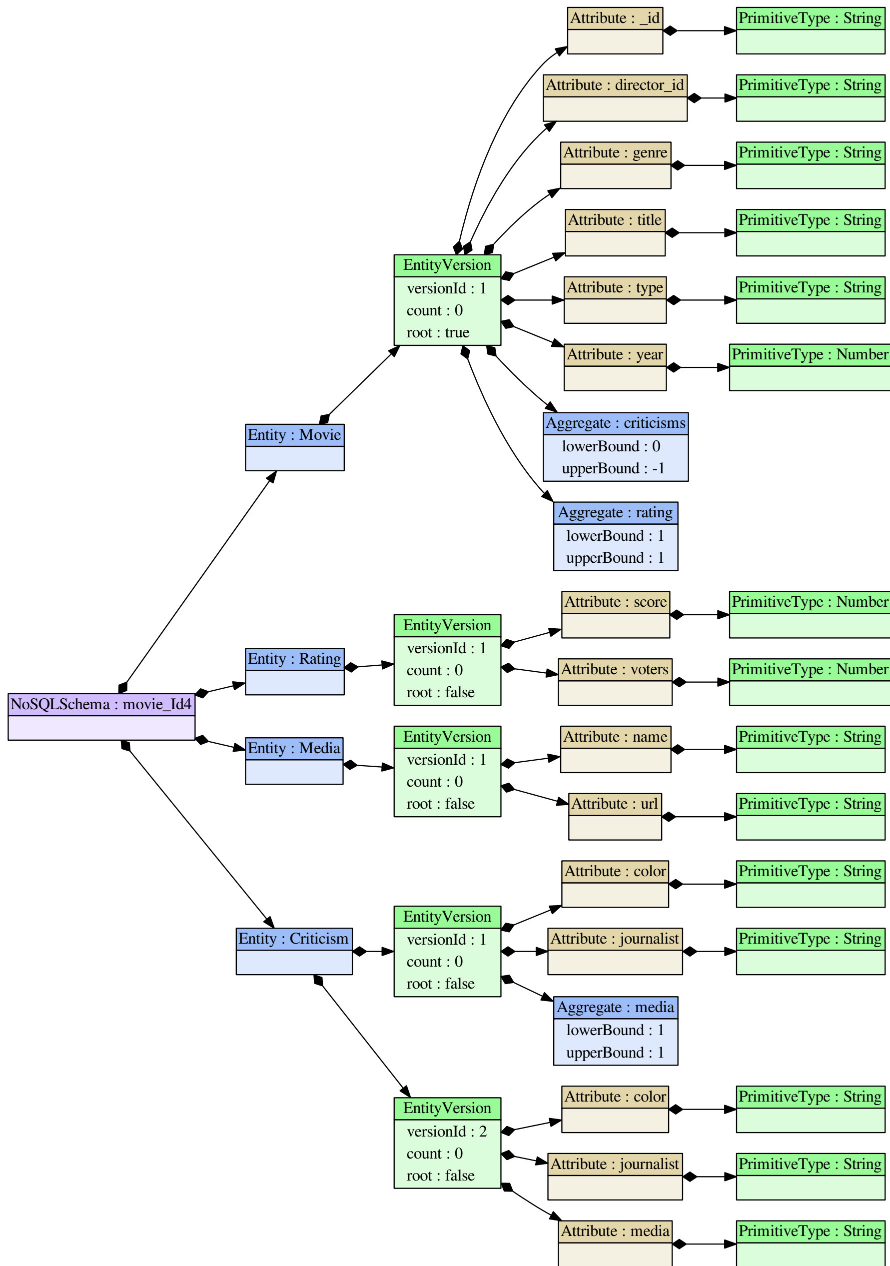

Figure 6.1: An excerpt of the diagram for Movies database generated with EMF to Graphviz.

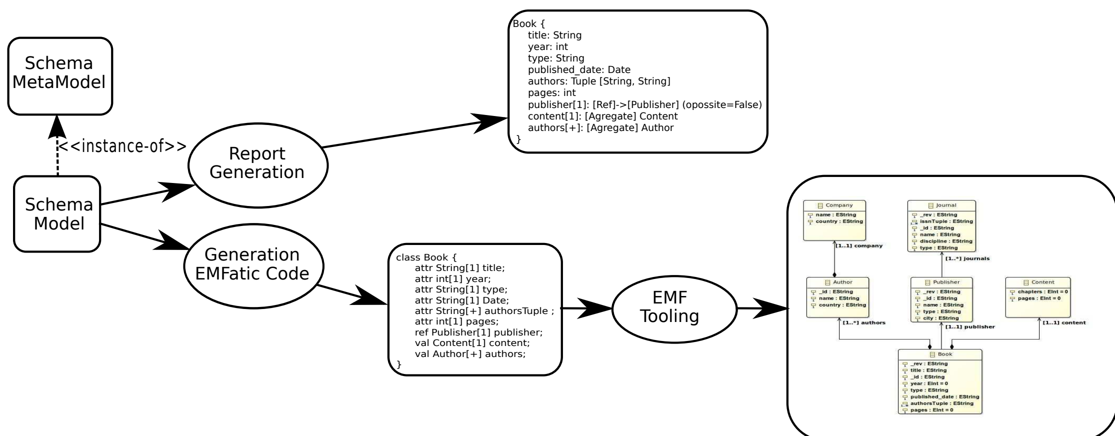

Figure 6.2: MDE solution to generate schema reports and diagrams

NoSQL-Schema models inferred are input to a model to text transformation that generates EMFatic code. EMFatic is a textual notation to express Ecore metamodels [46]. Our M2T transformation establishes a mapping between the NoSQL-Schema metamodel and the Ecore meta-metamodel shown in Figure 2.13 in Section 2.9.1, in a similar way to the mapping defined in [64] with the purpose explained in Section 3.1.5. These two mappings illustrate the benefits of representing models and metamodels uniformly. Next, we explain how we have mapped NoSQL-Schema metamodel elements to Ecore metamodels.

- Each entity generates a Ecore metaclass (*EClass* instance) of the same name; an Ecore attribute (*EAttribute* instance) is added to the metaclass for each entity's attribute, which has the same name and the types JSON are mapped to Ecore types (*EDataType* subclass instances): Number (integer) to EInt, Number (float) to EFloat, Boolean to EBoolean, String to EString, and JSON arrays of primitive type values to EList.

- Each aggregate and reference association of the NoSQL-Schema metamodel generates a composite and non-composite Ecore reference (*EReference* instance), respectively, with identical name and cardinality.

- If several versions of an aggregate or reference association exist in a model but with different cardinalities, the Ecore model generated will include only one version, and the cardinality resulting of applying the rules established for the UML package merge [92]: (i) the lower bound of the resulting multiplicity is the lesser of the lower bounds of

the multiplicities of involved relationships, and (ii) the upper bound of the resulting multiplicity is the greater of the upper bounds of the multiplicities of the matching elements.

The M2T transformation is executed to generate an EMFatic textual file, and the EMF/Eclipse tooling is used to generate the corresponding Ecore model which can be visualized as a UML class diagram by means of the metamodel editor integrated into EMF/Eclipse. Figure 6.3 shows the diagram for the running example of the Movies database.

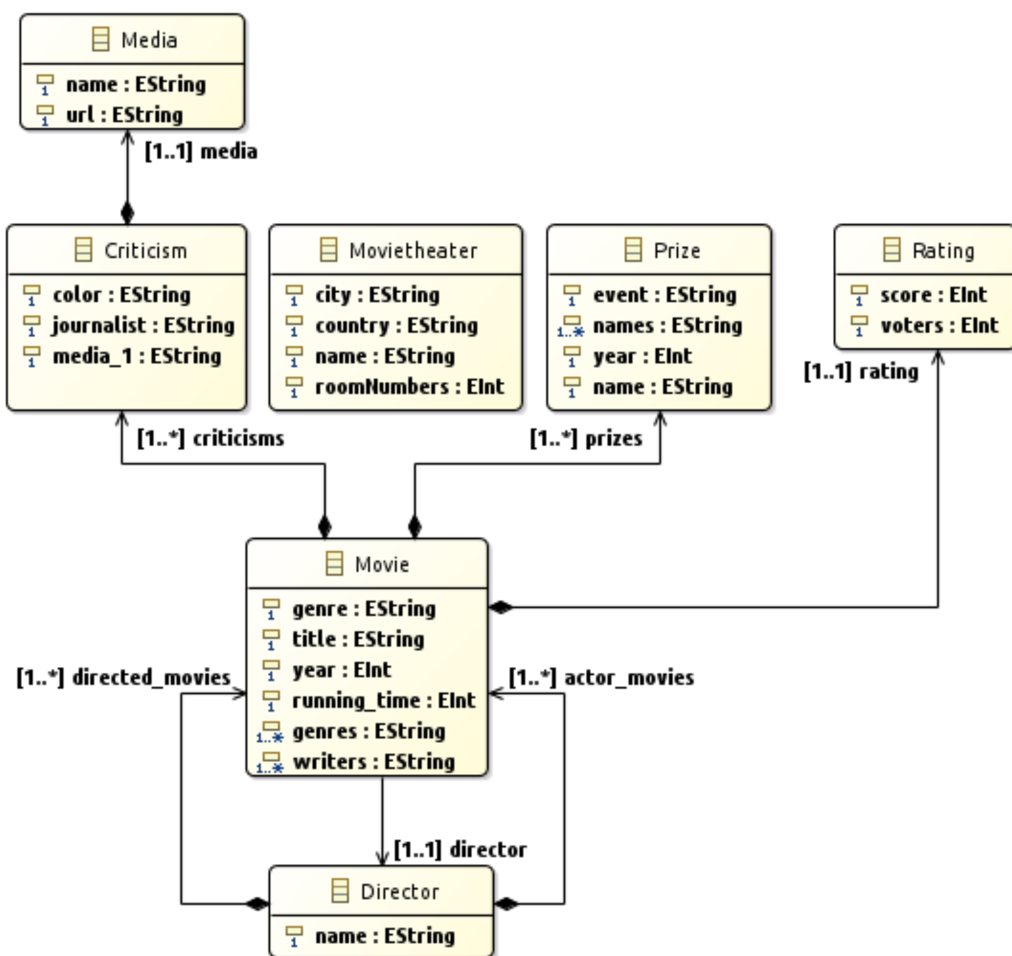

Figure 6.3: Entity Union Schema visualized for the running example database.

This diagram represent the entity database schema that includes all the entity union

schemas of the entities of the database. Recall that several strategies can be used to solve the problem of conflicting types when a field belongs to more than one version. In our inference process, the type of a field is the union of all the types encountered. In a class diagram, the representation of a union of type for the same field can not be visualized unless the name is changed. For instance, the *Criticism* entity contains the *media* attribute (String type) and the aggregation relationship of the same name (its association end is the *Media* entity). We have changed the name of the attribute to *media_1* in order to represent both properties.

Finally, noting that entity versions cannot be explicitly represented in UML class diagrams, but a new kind of representation is needed.

As shown in Figure 6.2, we also implemented an M2T transformation that automatically generates a textual report of all the entity versions existing in the database. This transformation also has the schema model inferred as its input. This was a proof of concept to illustrate that textual documentation (e.g. HTML code) could be generated from schema models, which could be complementary to the diagrams generated. Figure 6.4 shows a textual report of our running example. Attributes (name and type), and aggregate and reference relationships are indicated for each entity version.

6.3 Using PlantUML to visualize NoSQL Schemas

As indicated in Section 2.10, PlantUML [96] is a drawing tool for visualizing UML diagrams. It provides a textual DSL to express the diagrams. For instance, Figure 6.5 show the PlantUML code for the diagram shown in Figure 6.6. PlantUML code is transformed into .DOT code in order to be visualized by means of a Graphviz engine [55]. Using the PlantUML notation we can define formatting features of diagrams, such as colors and icons for elements. In addition, it is possible to have an attribute and association with the same name. This flexibility is not achieved in the previous solution based on generating Ecore models. We have used PlantUML to visualize three kind of diagrams: root version schemas, union entity schemas, and entity database schemas. Each kind of diagram is generated by means of a model-to-text transformation that generates the PlantUML code that corresponds to the input NoSQL-Schema model, as illustrated in Figure 6.5.

```
skinparam backgroundColor transparent
skinparam class {
```

```
Versions Entities:
Entity Movie {
 Version 1 {
  genre: String
  title: String
  year: int
  director[1]: [Ref]->[Director] (opposite=true)
  criticisms[+]: [Aggregate]Criticism
  prizes[+]: [Aggregate]Prize
 }
 Version 2 {
  genre: String
  title: String
  year: int
  running_time: int
  director[1]: [Ref]->[Director] (opposite=true)
 }
 Version 3 {
  genre: String
  title: String
  year: int
  director[1]: [Ref]->[Director] (opposite=true)
  prizes[+]: [Aggregate] Prize
 }
 Version 4 {
  genre: String
  title: String
  year: int
  rating[1]: [Aggregate] Rating
  director[1]: [Ref]->[Director] (opposite=true)
  criticisms[+]: [Aggregate] Criticism
 }
 Version 5 {
  genres: Tuple [String]
  title: String
  writers: Tuple [String]
  year: int
  director[1]: [Ref]->[Director] (opposite=true)
 }
}

Entity Movietheater {
 Version 1 {
  city: String
  country: String
  name: String
 }
 Version 2 {
  city: String
  country: String
  name: String
  roomNumbers: int
 }
}

Entity Media {
 Version 1 {
  name: String
  url: String
 }
}

Entity Rating {
 Version 1 {
  score: int
   voters: int
 }
}

Entity Director {
 Version 1 {
  actor_movies[+]: [Ref]->[Movie] (opposite=true)
  directed_movies[+]: [Ref]->[Movie] (opposite=true)
  name: String
 }
 Version 2 {
  directed_movies[+]: [Ref]->[Movie] (opposite=true)
  name: String
 }
}

Entity Criticism {
 Version 1 {
  color: String
  journalist: String
  media[1]: [Aggregate] Media
 }
 Version 2 {
  color: String
  journalist: String
  media: String
 }
}

Entity Prize {
 Version 1 {
  event: String
  names: Tuple [String]
  year: int
 }
 Version 2 {
  event: String
  name: String
  year: int
 }
}
```

Figure 6.4: Textual report of entity versions in the running example database.

```
  BackgroundColor Blue \n
  ArrowColor Blue
  BorderColor Red \n
  FontSize 18 \n
  FontName Arial \n
}

skinparam stereotypeCBackgroundColor Blue
skinparam stereotypeCBorderColor SpringGreen

Class Movie4<<(R,Turquoise)>>{
        <b> String _id
        <b> String genre
```

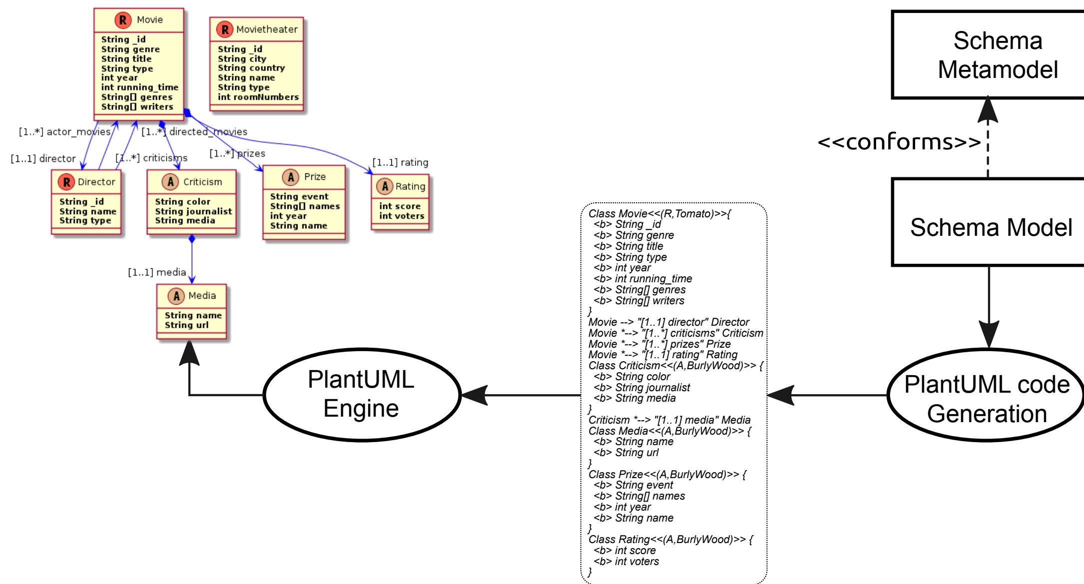

Figure 6.5: Visualization of schema diagrams by using PlantUML.

```
        <b> String title
        <b> String type
        <b> int year
}

Movie4 --> "[1..1] director" Director

Class Director<<(E,Tomato)>>{
        <b> String _id
        <b> String name
        <b> String type
        <i><color:Navy>ref Movie[] actor_movies</color>
        <i><color:Navy>ref Movie[] directed_movies</color>
}

Movie4 *--> "[1..1] criticism1" Criticism1

Movie4 *--> "[1..1] criticism2" Criticism2


Class Criticism1<<(V,BurlyWood)>>{
        <b> String color
        <b> String journalist
}

Criticism1 *--> "[1..1] media1" Media1


Class Media1<<(V,BurlyWood)>>{
        <b> String name
        <b> String url
}


Class Criticism2<<(V,BurlyWood)>>{
        <b> String color
        <b> String journalist
        <b> String media
```

```
}

Movie4 *--> "[1..1] rating1" Rating1

Class Rating1<<(V,BurlyWood)>>{
        <b> int score
        <b> int voters
}

@enduml
```

Root Version Schemas As explained in Section 4.1, a *version schema* represents the type (schema) of a particular entity version and expresses the type for each property that can be: primitive, or a referenced or embedded entity, or an array. Such schemas can be visualized by showing only the entity version directly referenced or either the complete graph formed by including the indirectly referenced entity versions. We focused here on the representation of version schemas for root entities, and its schema tree is shown as a UML class diagram. For instance, Figure 6.6 shows in form of UML class diagram the version schema tree shown in Figure 4.4 for the *Movie_4* entity version. Each entity version in the schema is represented as a class, and a letter within a small circle is used to distinguish the root entity ("R") ("V") of the embedded entity versions. The class name is the entity version name in the schema model (i.e. property name followed by an integer identifier from 1 to *N* that is used as version number). All the entity versions which are directly or indirectly nested to the root entity version are shown by means of unidirectional composite relationships whose name and cardinality are the same than the corresponding aggregation elements of the schema model. When an entity version named *v1* aggregates an array of *N* entity versions named *v2*, this is represented by means of *N* unidirectional composite relationship from *v1* to *v2*, whose cardinality is 1..1 and the role name is the name of the corresponding schema model property. As explained in Section 5.2, the target of a reference is an entity not a entity version, therefore a version schema can also include entities, and the "E" letter is used to label the classes that represent entities. For entities, the diagram shows the embedded or referenced entities but not entity versions. If an entity references to the root entity version then the references links are not shown, but the specification is enclosed after the attribute list in the format: keyword *ref* is followed of the entity name and the property name.

Figure 6.6 shows the diagram for the schema of the *Movie4* entity version. As observed,

Movie4 aggregates an array that includes objects of two versions of the *Criticism* entity, and one of them, in turn, aggregates a *Media1* entity version. *Movie4* also aggregates a *Rating1* entity version, and references to the *Director* entity. Note that the diagram does not include the link for references from *Director* to *Movie* but they are represented in the format commented above: *refMovie*[]*actor_movies* and *refMovie*[]*directed_movies*.

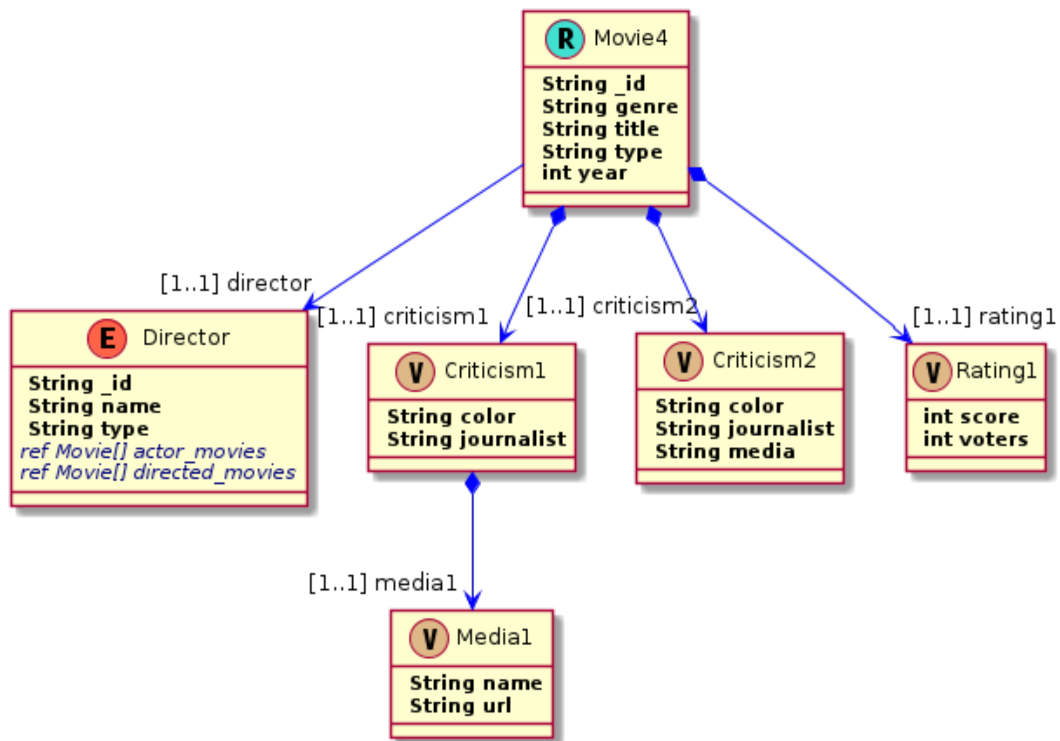

Figure 6.6: Root Entity Version Schema for *Movie_4* generated with PlantUML.

The root version schema for *Director2* is shown in Figure 6.7. *Director2* references to the *Movie* entity (*directed_movies* property), therefore the diagram includes the union schema for this entity. That is, *Movie* aggregates to *Criticism*, *Prize*, and *Rating* entities, references to *Director* (*ref Director director*), and has a attribute for each entity's attribute with the limitations commented above for UML class diagrams: there not exist two attributes with identical name but different type, although an attribute and relationship can have identical name for the same entity.

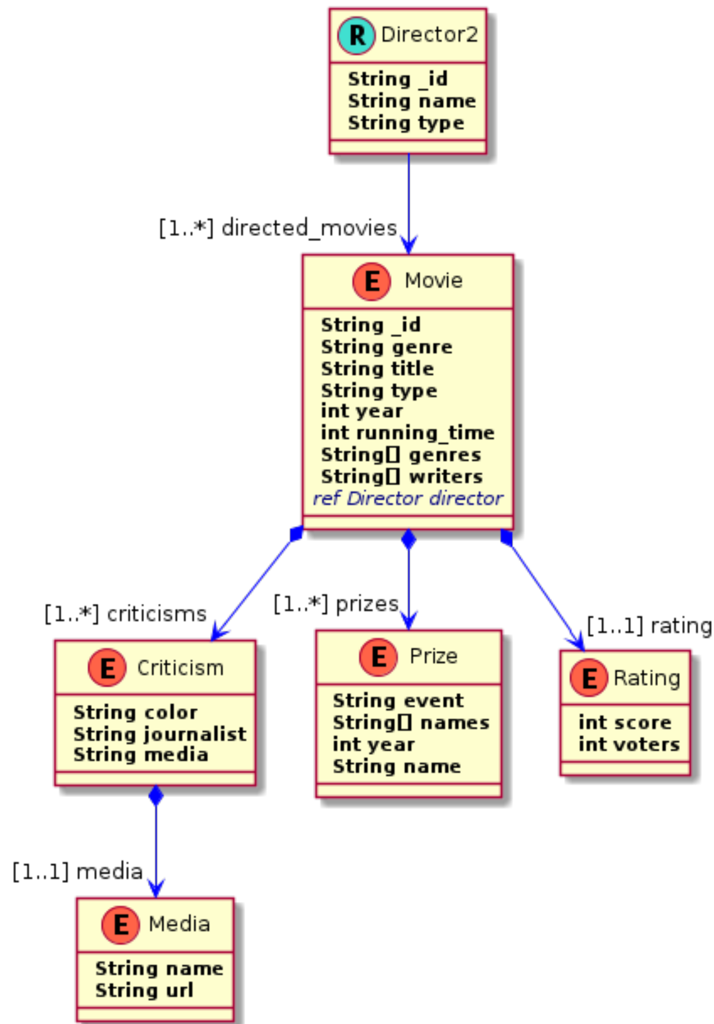

Figure 6.7: Root Entity Version Schema for *Director_2* generated with PlantUML.

Entity Union Schemas As explained in Section 4.1, an *entity schema* contains the set of schemas of the different entity versions of that particular entity, and an entity union schema is formed by joining all the properties contained in the schemas of the entity versions of the entity. Therefore, the definition of an entity union schema includes primitive types, arrays, and the union schemas that correspond to the embedded or referenced entities. When the schema defines a root entity then it is called *root union schema*. Next, we describe how such schemas could be represented as class diagrams by using PlantUML. We have followed the

strategy explained above for visualizing entities in root version schemas. When several version schemas have a property with identical name but the type is different, the union type inferred for this property can only be visualized if it is satisfied that it has only two versions: one includes the property with a primitive type (or a tuple) and the other one is a relationship, but the rest of possible unions of types would cause an error because would have several attributes (or relationships) with the same name and different type (or association end). Figure 6.8 shows the union schema for the *Movie* entity, which includes eight attributes (two of them are tuples of String), three composite relationships for the *Criticism*, *Prize* and *Rating* union schemas, and a reference relationship to the *Director* entity union schema. The diagram also shows the relationships of the aggregated or referenced union schemas, for instance, the composite relationship from *Criticism* to *Media* and two reference relationships from *Director* to *Movie*. While the previous diagrams show all the referenced or embedded entity versions involved in the definition of a version schema for a root entity, the diagram for an entity union schema only includes entities. Therefore, in the diagram of Figure 6.8 the references relationships from *Director* to *Movie* are shown because no confusion is caused. Note that *Criticism* entity contains the *media* attribute of type String along and the aggregation relationship of the same name whose association end is the *Media* entity.

Entity Database Schemas As explained in Section 4.1, an *entity database schema* is formed by the the set of entity union schemas that describe all the root entities of the database. It is worth to note that an entity union schema can be direct or indirectly connected to the rest of entity union schemas existing in the database, and then such an schema will be equivalent to the entity database schema. Therefore, a diagram of the entity database schema is formed by superposing all the diagrams for root union schemas. Figure 6.9 shows the diagram for the entity database schema of our running example. This diagram is the same as that shown in Figure 6.9 but including the union schema for *MovieTheater* that is not involved in the union schema for *Movie*. In this case, *MovieTheater* is not connected to the rest of schemas in the diagram. This could be explained by some facts as that (i) *MovieTheater* objects have been registered but they should be connected to other entities, e.g. Movies, or (ii) the database is being migrated and references to *MovieTheater* objects from other objects have been removed, or (iii) Movies and MovieTheater are managed by different ap-

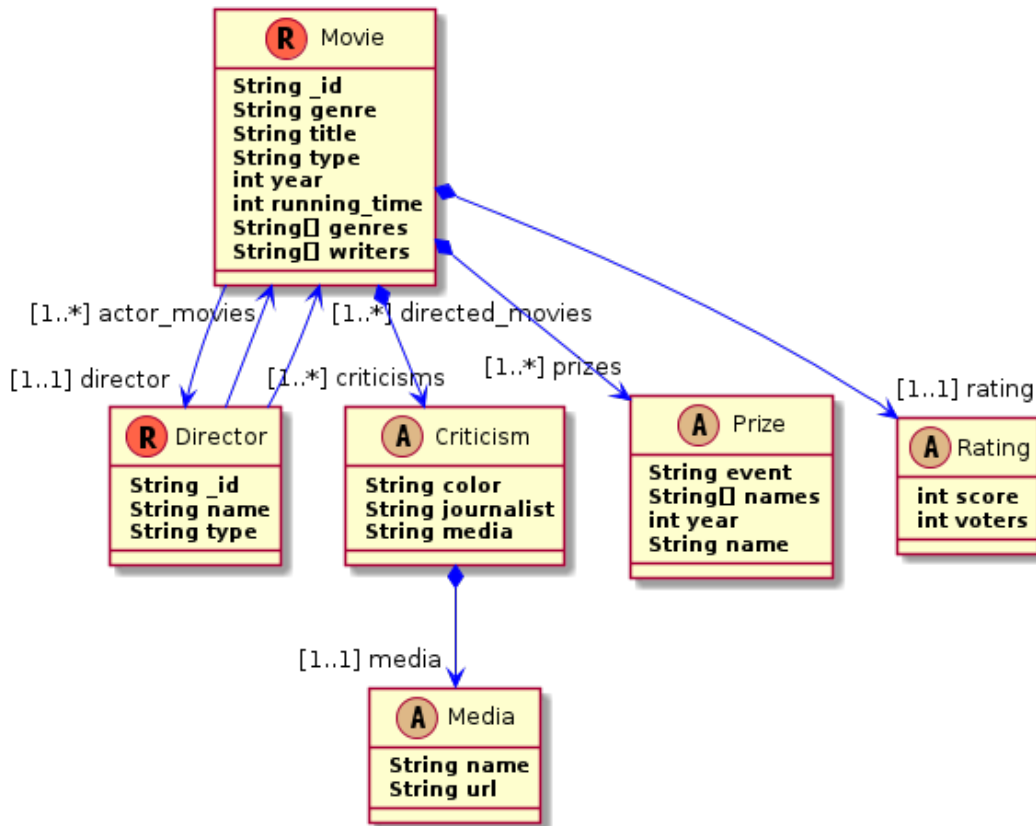

Figure 6.8: Entity union schema for Movie generated with PlantUML.

plications.

6.4 Using Sirius to Visualize NoSQL Schemas

The previously described solutions to visualize NoSQL schemas evidenced the convenience of defining a specific notation for this purpose because UML class diagrams are not appropriate to represent entity versions. For this aim, a diagramming tool has been developed as part of a Master's Thesis [29, 60]. The applied strategy has been to create a model editor by means of Sirius [7]. This solution takes advantage of the fact that our inference process generates models that are instances of a Ecore metamodel. Sirius is a robust and powerful tool to define concrete syntax for an existing metamodel. The development time and effort to achieve the schema visualization is considerably reduced by using this kind of solution in

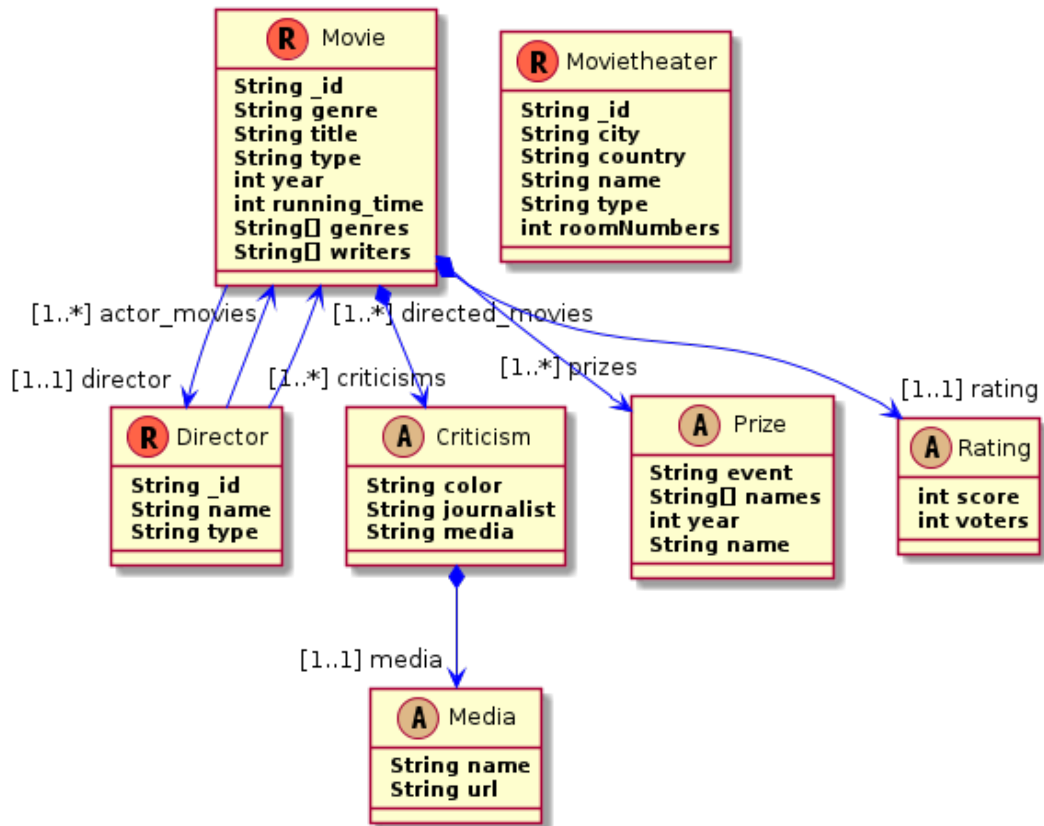

Figure 6.9: Entity union schema for Movie generated with PlantUML.

relation to create the editor from scratch. Figure 6.10 shows the visualization process. First, we define the notation for the Schema metamodel by using the capabilities offered by Sirius for this task. Taking as input the metamodel and its notation, Sirius generates (i) an editor to create and visualize diagrams of models that conform to the input metamodel, and (ii) a model injector that generates an EMF model from the graphical representation. Next, we will comment the main features of the different diagrams and views created for our metamodel. [29, 60].

Global View Tree This view shows a tree with three branches that are labeled as *Schema*, *Inverted Index* and *Entities*, as illustrated in Figure 6.11. *Schema* labels the list of all the root entities with their version schema; given a root version schema, the user can browse their embedded and referenced schemas. For instance, Figure 6.11 shows the aggregated and ref-

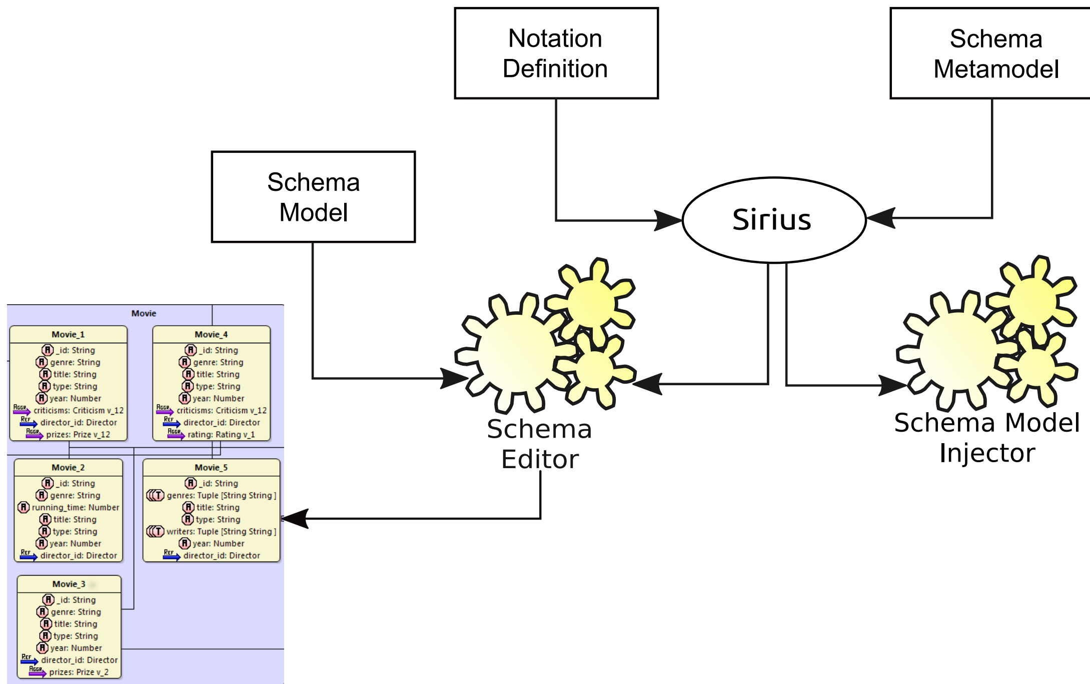

Figure 6.10: Visualization of schema diagrams by using Sirius.

erenced schemas from the root version schemas of *Movie*, *Director* and *MovieTheater*. Schemas for *Director_1* are shown in that figure. *Inverted Index* labels an inverse index of versions. This kind of index has been defined to navigate from a root or embedded version schema to all the root version schemas from which is referenced (for example, *Director_1* is referenced from *Movie_1*). *Entities* labels a list of all the entities that exist in the database. Both root and embedded entities are included in the list. The user can select an entity to display its entity versions, and then she/he can inspect their properties and types. Therefore, this *Entities* branch shows the *database schema* as defined in Section 4.1.

In this tree, each kind of element is identified by means of square or circle icons in order to ease the understanding of the schema to the user. The following icons have been designed.

- Entities are represented with a pale purple icon that encloses the tag "E" and the entity name.
- Entity versions are represented with a yellow icon that encloses the "EV" tag and the

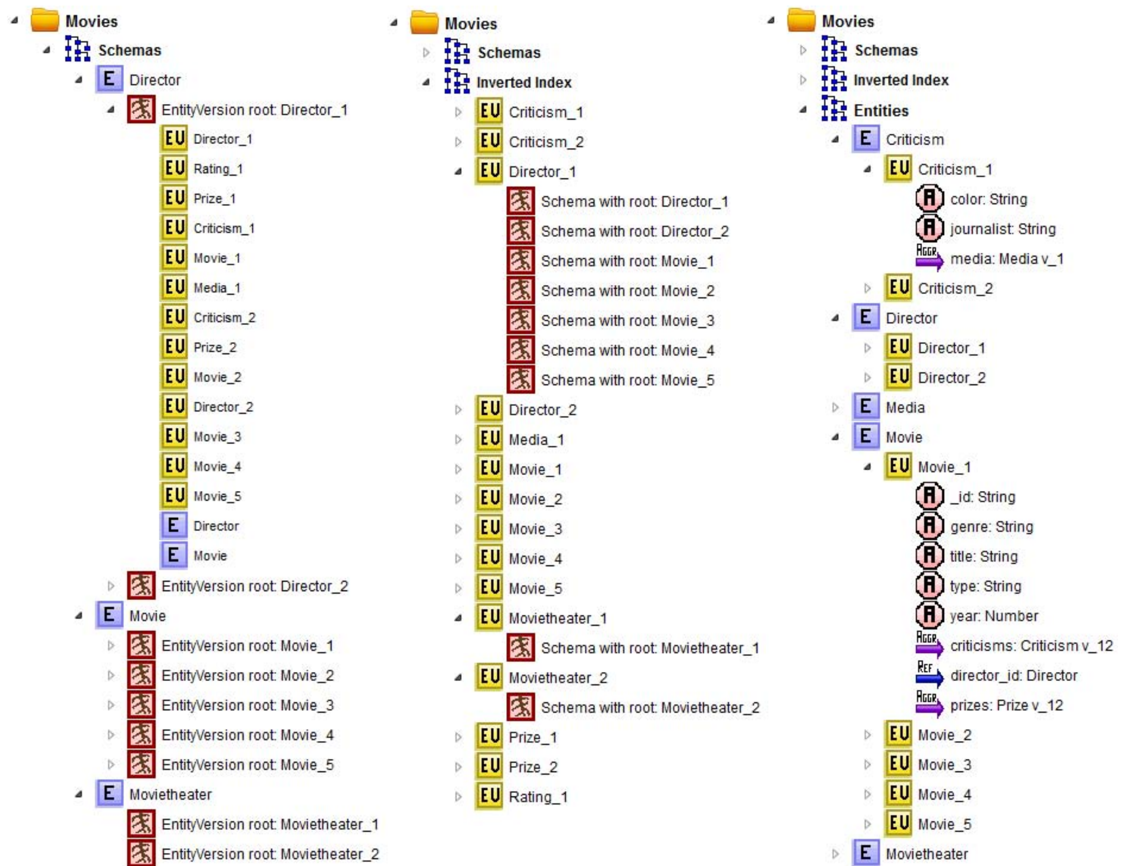

Figure 6.11: Global Schema Tree for the *Movie* example.

entity version name.

- Root entities are represented with a root picture, and the name of the root entity version.

- Aggregate and reference relationships are represented by means of an arrow icon that has dark blue and purple colors, respectively. The icon is followed by the relationship name and and the target entity name.

- Primitive attributes and tuples are represented with a pink icon that enclosed the "A" tag and the attribute and type names.

These icons are used in all the diagrams created to provide a uniform view to user. It is

possible to navigate from the Global View Tree to the other diagrams by means of contextual menus.

Database Schema Diagrams A *database schema diagram* shows the information of the *Entities* branch in similar form to a UML class diagram, as observed in Figure 6.12. With this diagram, the user can see at a glance what are (i) the database entities, (ii) the set of version schemas of each entity, (iii) the attributes and relationships of each version schema. Aggregation and reference relationships are visualized as a solid line and the *aggregates* and *references* tags are used to differentiate between them.

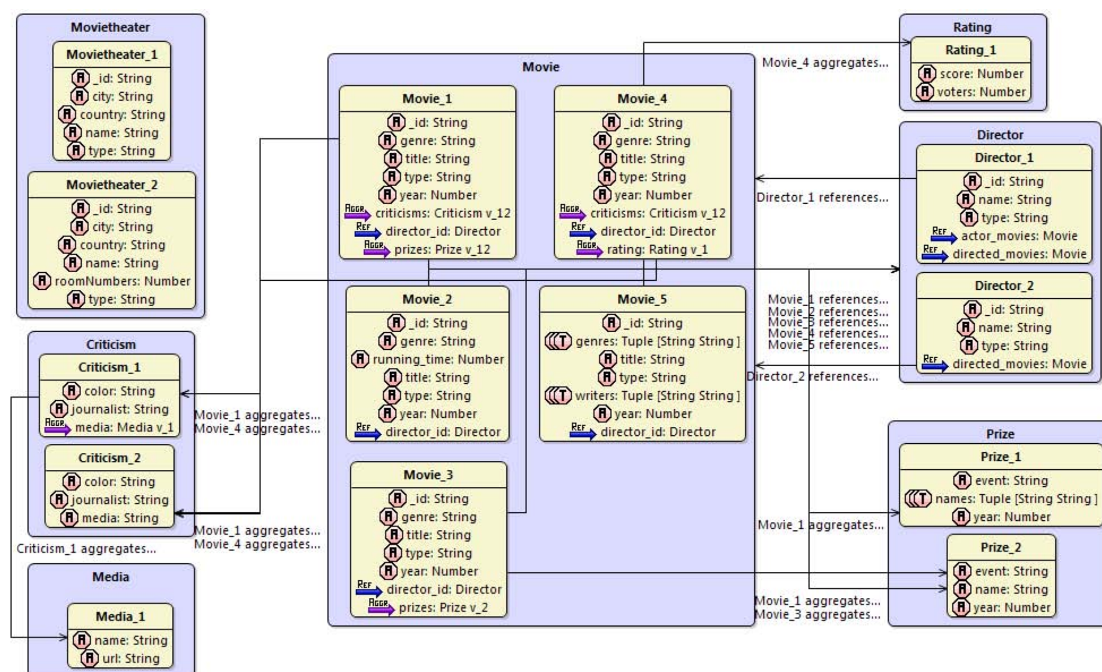

Figure 6.12: Database schema diagram for *Movies* database.

From these diagrams it is possible to access to the two diagrams defined to represent root version schemas: *plain* and *nested version schemas*, and the contextual menu of an entity can be used to navigate to its *entity diagram*.

Root Version Schema Diagrams A root version schema diagram represents a root version schema as defined in Section 4.1. Plain and nested diagrams differ in how the direct relationships between version schemas are represented. In both diagrams, schemas are represented

as rounded rectangles. As shown in Figure 6.13, these relationships are indicated by arrows in plain diagrams just like they appear in a database schema diagram. Instead, the embedded schemas are visually represented as rectangles nested to the rectangle of the root schema in a nested diagram, as shown in Figure 6.14,

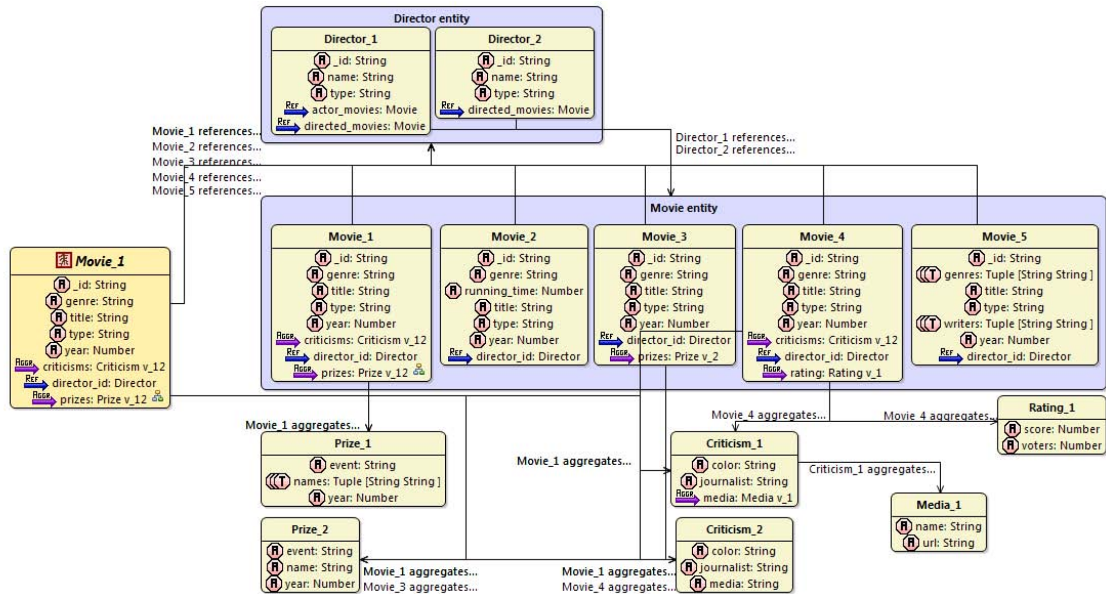

Figure 6.13: Plain version schema diagram for *Movies* database.

A nested diagram offers a more compact view and more clearly shows the nested level for each root schema. Instead, a plain diagram highlights the relationships between version schemas.

Entity Schema Diagrams An entity schema diagram represents an entity schema as defined in Section 4.1. They show the version schemas as rectangles nested into a rectangle that represents the root or embedded entity it belongs to, as shown in Figure 6.15. In this diagram, the *Criticism* entity has two versions and the corresponding version schemas are shown, for instance *Criticism_1* aggregates a version schema of the *Media* entity.

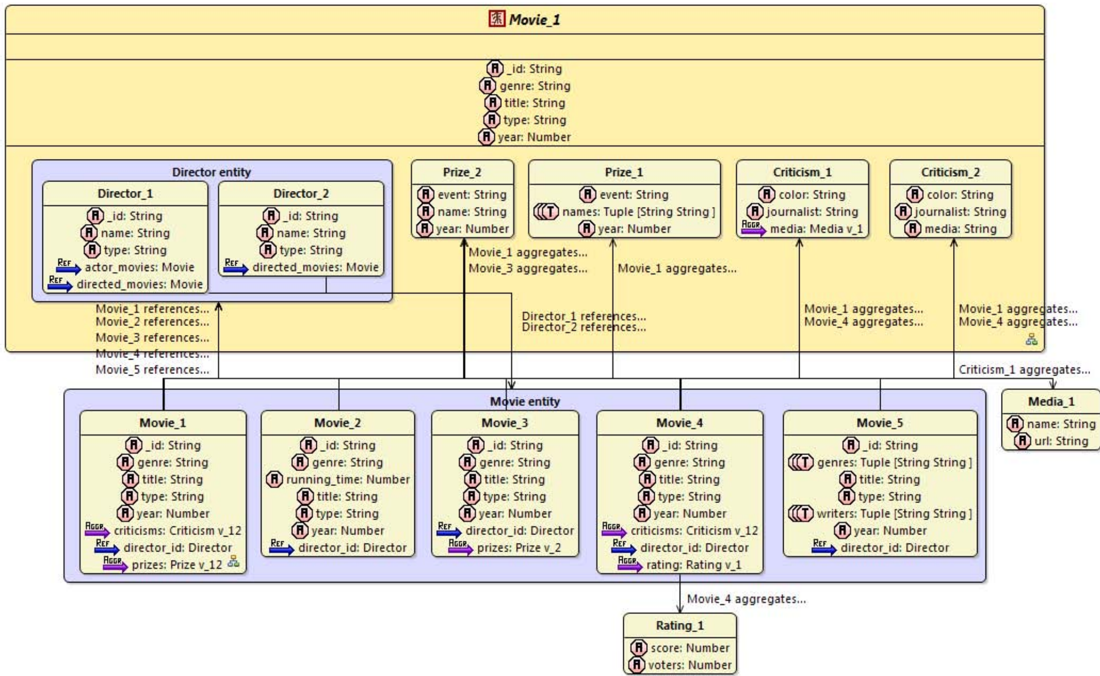

Figure 6.14: Nested version schema diagram for *Movies* database.

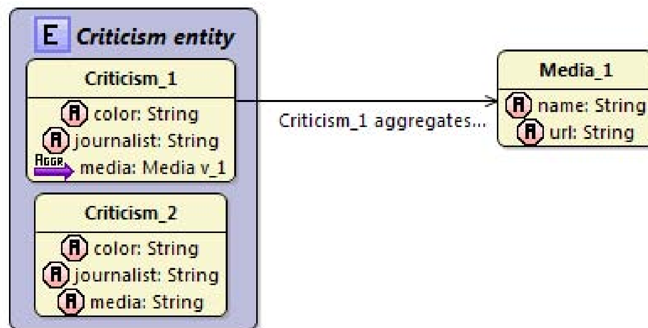

Figure 6.15: Entity Schema Diagram for *Criticism*

.

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

Alan Kay

7 Code Generation from Schema Models

In this chapter, we will show how the inferred schemas can be used to generate code for NoSQL database applications. We will describe the MDE solutions developed to automatically generate (i) code for ODM mappers, (ii) data validation code, and (iii) code to classify objects into entity versions.

7.1 Generating code for ODM Mappers

An MDE solution has been designed and implemented to automate the usage of ODM mappers when the database already exists. We have considered Mongoose [81] for MongoDB[80], but the solution presented is applicable to other object-document mappers.

We shall first present an overview of the proposed MDE solution. Next, we will introduce the *EntityDifferentiation* metamodel and explain in detail the two stages of the model transformation chain that implements the solution: (i) how *NoSQL-Schema* models are transformed into *EntityDifferentiation* models. and (ii) the generation of Mongoose schemas from *EntityDifferentiation* models.

7.1.1 Overview of the approach

We have defined a two-step model transformation chain which has as input an inferred *NoSQL_Schema* model, and generates Javascript code of Mongoose artefacts. The generated artefacts are mainly the database schema and validators. Figure 7.1 shows this generation process, which will be explained in detail in the next sections. The first step of the chain is a M2M transformation that reorganizes the information included in a NoSQL-Schema in a way that facilitates the code generation. This transformation generates a model that conforms to the *EntityDifferentiation* metamodel which will be explained later in this section. The second step is a M2T transformation that generates Mongoose code from the model obtained in the previous step. The *EntityDifferentiation* metamodel has been defined to make writing the M2T transformation easier. The M2M transformation has been implemented in Java, while the M2T transformation has been written in Xtend [2].

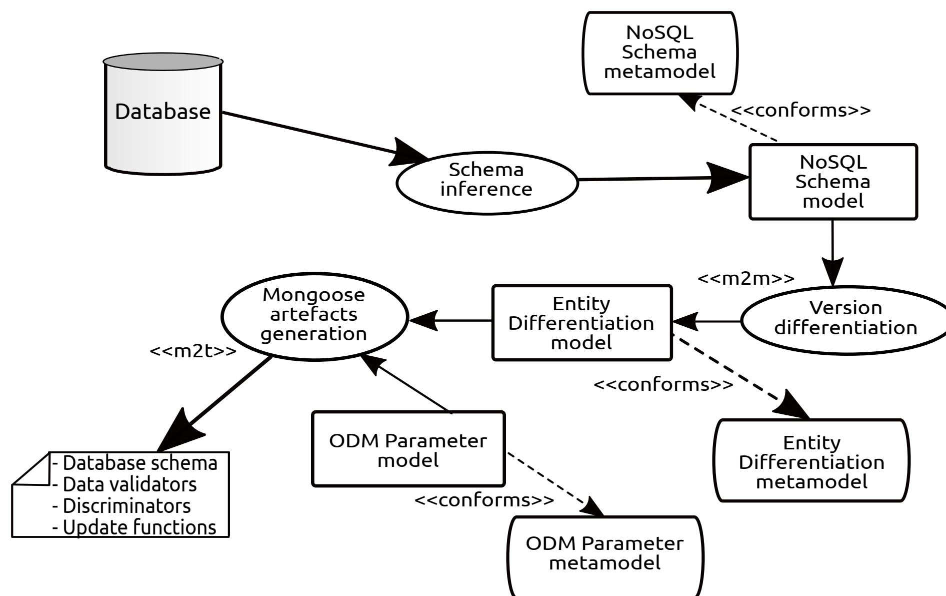

Figure 7.1: Overview of the Proposed MDE Solution.

Our solution deals with the existence of more than one version for data entities, which imposes an additional complexity; conversely, it captures all the variability in the data base. We have considered that the fields of an entity version can be of three kinds: (i) *Common* to all entity versions (i.e. they are part of the all documents of the entity); (ii) *Shared* with other entity versions; and (iii) *Specific* to particular entity version. For instance, in our run-

ning example, the fields *title*, *year*, *director* and *genre* are common to all Movie objects, while the field *prizes* is shared by two versions of Movie, and the field *rating* is specific to the version defined for the Movie object with _id = 4.

7.1.2 Generating EntityDifferentiation Models

Generating artifacts to manage the different entities and entity versions often has to differentiate between the properties common to all entity versions and other properties specific of a given entity version, as indicated above.

For instance, once a document of the database is obtained, in order for it to be classified as belonging to a given entity version, some *property checks* must be performed. These checks can ignore the common properties of all entity versions of a given entity, and take into account those properties that differentiate each entity version of the rest.

This may seem a trivial task, but some subtleties that will be addressed in this section made it easier to take this process as separate of the artifact generation process, and also made the generation process itself easier. Thus, the *Entity Differentiation Metamodel* was created with the main purpose of distinguishing between common and specific properties of an entity version.

This metamodel is shown in Figure 7.2. The root element of this metamodel is *EntityDifferentiation* that aggregates a set of elements (*EntityDiffSpec*) that specify the differences between the versions of an entity. An *EntityDiffSpec* aggregates a set of specification of properties common to all entity versions (*PropertySpec*) and a set of properties specific of a given entity (*EntityVersionProp*). A *PropertySpec* has the *needsTypeCheck* boolean attribute whose purpose is explained below, and an *EntityVersionProp* aggregates a set of *PropertySpec*s and a set of *PropertySpec*s that do *not* have to be in a specific entity version (also explained below). Note that the *Entity Differentiation Metamodel* has references to the elements of the NoSQLSchema metamodel: (i) an *EntityDiffSpec* references to an *Entity*, (ii) an *EntityVersionProp* to an *EntityVersion*, and (iii) a *PropertySpec* to a *Property*. Actually, an *EntityDifferentiation* model organizes information contained in a Schema model in a way convenient to distinguish between common and specific properties in entity versions, and it could be useful for other applications.

The rationale behind this metamodel is based in two facts of the inference process:

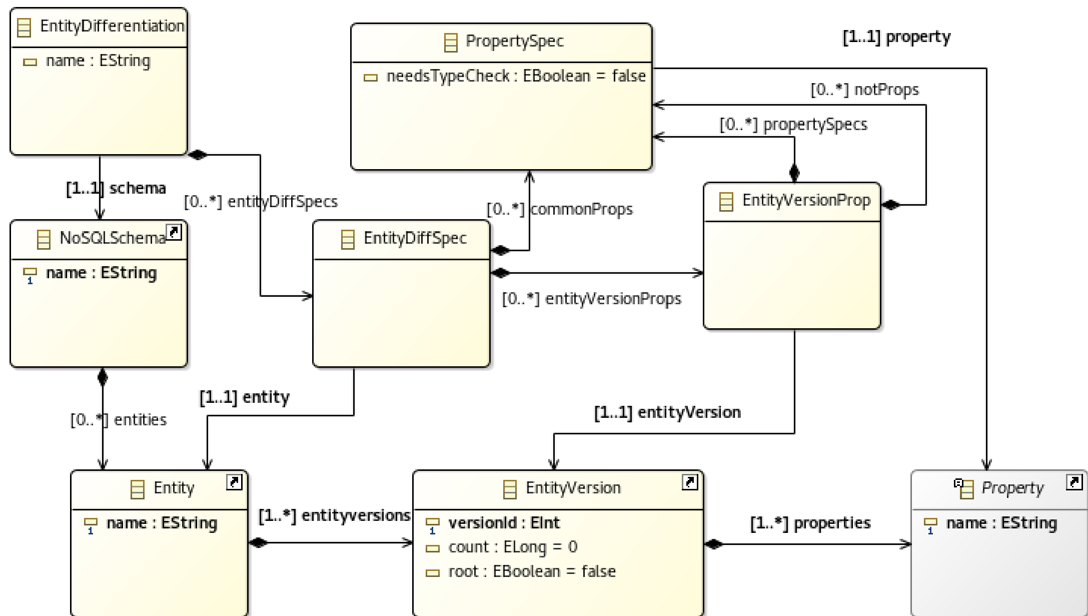

Figure 7.2: Entity Differentiation Metamodel.

1. The inference process is *complete*, that is, all documents in the database are considered, and their different entity versions recorded. Each document of the database belongs to *exactly one* entity version.

2. In order to differentiate between versions of a given entity, only properties specific of the given entity version need to be considered.

Instances of this metamodel are obtained via a model-to-model transformation from the *NoSQLSchema* metamodel. For each *Entity*, an *EntityDiffSpec* model element is generated. As said above, this element holds a set of common properties across all the versions (*commonProps*), and a set of differentiation properties for each version (*entityVersionProps*), These sets are obtained as follows. Common properties are those properties that are present (with the same name and type) in all the entity versions of a given entity. Conversely, the set of specific properties for a given entity version is composed of the properties that are present in this entity version, but are not present in all other entity versions.

Properties in a *EntityDifferentiation* model are linked through the *PropertySpec* class. This class includes a *needsTypeCheck* attribute to signal when a discrimination cannot be

made just using the *name* of the property. For instance, if two entity versions share a property with the same name but with different type, the fact that a document has a property with that name cannot be used to discriminate between these two entity versions: a type check must be performed. So, the *needsTypeCheck* attribute is set for the properties that appear in any other entity version with the same name *but with different type*.

An excerpt of the generated model for the database example can be seen in the Figure 7.3.

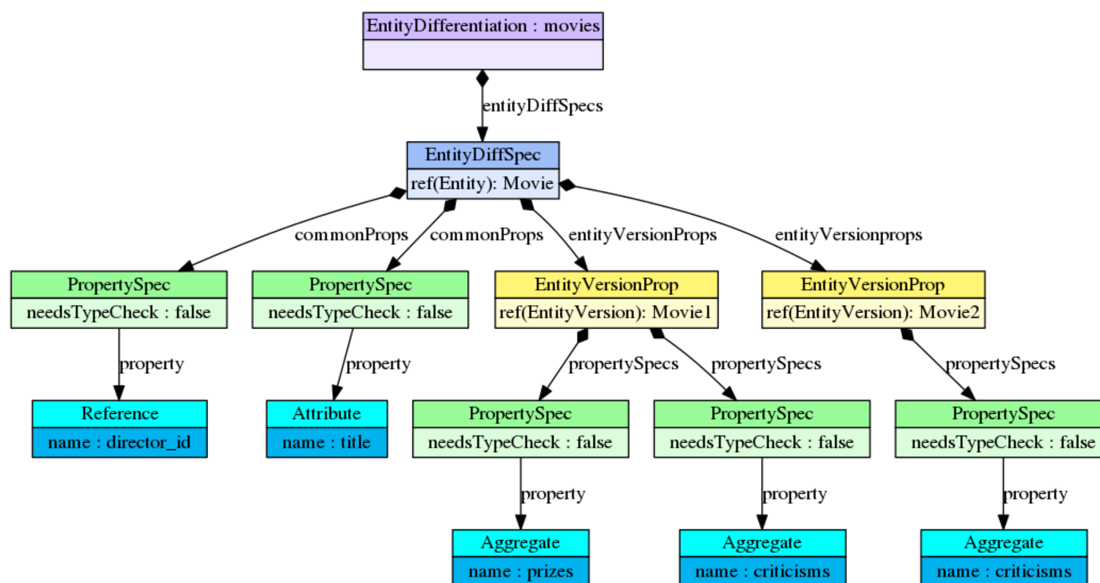

Figure 7.3: Excerpt of the EntityDifferentiation Model for the Example.

7.1.3 Generating Mongoose Schemas

A Mongoose schema defines the structure of stored data into a MongoDB collection. In document databases, such as MongoDB, there is a collection for each root entity. In our database example there would be two collections: *Movie* and *Director*. Therefore, a Mongoose schema should be defined for each of the two collections. Such schemas are the key element of Mongoose, and other mechanisms are defined based on them, such as validators, discriminators, or index building. In this Section, we shall explain the process of generating schemas by means of the m2t transformation indicated in Section 7.1.1. Figure 7.4 shows the generated schemas for the example database (*MovieTheather* is not considered).

```
// Movie Schema
var criticismsSchema = {
  color: {type: String,
          enum:['green', 'yellow',
                'red'],
          required: true},
  journalist:{type: String, unique:true,
              required: true},
  media:{type: String, required: true},
  url: String
}
var prizesSchema = {
  event:{type: String, required: true},
  names:{type: [String], required: true},
  year: {type: Number, required: true}
}
var movieSchema = new mongoose.Schema({
  title:{type:String, maxlength:40,
         unique:true, required:true},
  _id:  {type:String, index:true,
         required:true},
  year: {type:Number, index:true,
         required:true},
  type: {type:String, required:true},
  director_id: {type: String,
                required: true,
                ref:'Director'},
  genre: {type:String,
          enum:['drama','comedy',
                'children'],
          required:true},
  criticisms: {type:criticismsSchema},
  prizes: {type:prizesSchema}
},{collection:'Movie'});

var Movie = mongoose.model('Movie',movieSchema);

// add Director1 schema referenced by Movie1
var directorSchema = new mongoose.Schema({
  _id:         {type:String, index:true,
                required:true},
  name:        {type:String, unique: true,
                required:true},
  type:        {type:String, required:true},
  actor_movies:        {type:String,
                        ref:'Movie'},
  directed_movies: {type:String,
                    required:true,
                    ref:'Movie'}
},{collection:'Director'});

// add for Director 1 entity Version
directorSchema.path('actor_movies').required();

var Director = mongoose.model('Director',
                               directorSchema);
```

Figure 7.4: Generated Mongoose Schema.

Aggregations and references can be specified in Mongoose schemas. An aggregation is expressed as an nested document which defines the schema of the aggregated entity. A reference is expressed by means of the *ref* option in the definition of the type of an attribute. In addition to the type (i.e. a primitive type as `ObjectID`, `Number` or `String`), the *ref* option is used to indicate the name of a model of the referenced schema. In Figure 7.4, the *Movie* schema aggregates schemas for *Prize* (*prizes* field) and *Criticism* (*criticisms* field), and includes a reference to *Director* documents stored in the *Director* collection (*director_id* field). The *ref* option is used by Mongoose to ease the use of references in queries.

Note that we are assuming a scenario in which a company wants to use Mongoose for an existing database in order to take advantage of its facilities to check that data are correctly managed. Then, our tool would infer the database schema and generate code facilitating the use of the mapper. In generating schemas, we had to consider the existence of entity versions. For this, we have used the *require* validator that Mongoose provides to specify that

a value for a particular field must always be given to save documents of a schema. Specifications of fields that are common to all the versions of an entity includes the validator *requires:true* to guarantee that any document stored of the entity will include these attributes. To work with a particular entity version, developers should add to the schema *require* restrictions for each of the specific fields of the version. In the schema of Figure 7.4, the *Movie* schema includes the *require* for the four common fields.

The transformation works as follows to generate database schemas: A schema is generated for each *EntityDiffSpec* connected to a root entity. For each of them, its common and entity version properties are added to the generated schema, but the *require* option is added only for common properties. For each aggregate property, an external declaration of type is added to improve the legibility of the schema. A model is created for schemas referenced from other schemas, which is needed to add the *ref* option in the declaration of the reference property. Note that this strategy will recursively operate because the existence of aggregate and reference properties. Therefore, we generate the entity database schema.

7.2 Generating other Mongoose Artefacts

In addition to the schema definitions and the management of references between documents, Mongoose provides functionality to facilitate the development of MongoDB applications, such as validators, discriminators, and index specification. To automatically generate Mongoose code involved in all these mechanisms, we have created a domain-specific language (DSL) aimed to specify the information needed for such generation. This DSL is named *ODM Parameter Language* and it is independent of a concrete mapper technology. Figure 7.5 shows an example of specification for the entities of our database example. This DSL has been created with Xtext [122], and models are obtained by means of the parser generated by this tool. These DSL models are input to the m2t transformation that generates Mongoose artefacts from *EntityDifferentiation* models, as shown in Figure 7.1.

In Mongoose, the validation is defined at the schema level. Some frequently used validators are already built-in. The *require* and *unique* validators can be applied to any property. As explained in previous Section, we have used *require* to specify what properties are common to all the versions. The *unique* validator is used to express that all the documents of a collection must have a different value for a field of primitive type. Other examples of val-

```
ODMParameters {
    mapper: Mongoose
    Entity Movie{
        validators{(genre : 'enum (drama,comedy, children)'),
                   (title : 'length < 40')
        }
        uniques (title)
        updates (genre, title)
        indexes (_id -> kind:Hash, year -> kind:Sorted)
    }

    Entity Director{
        uniques(name)
        updates(name)
        indexes (_id -> kind:Hash)
    }
    Entity Criticism{
        validators{(color : 'enum (green,red, yellow)')}
        uniques (journalist)
    }
    discriminator Entity MovieTheater()
}
```

Figure 7.5: Specification Example with the ODM Parameter Language.

idators are *min* and *max* for Number fields, and *enum*, *minlength* and *maxlength* for String fields. Indexes are also defined at schema level, for instance an index can be specified with the *index* option or the *unique* validator (which also implies the creation of an index). Examples of use of these validators are shown in the schema in Figure 7.4. For instance an enumeration is defined for the *color* field of *Criticism* and the *title* field of *Movie* is *unique*. These validators have been generated from the information provided by the DSL specification shown in Figure 7.5.

Our schema inference mechanism cannot discover the decisions behind a version entity. As indicated in Section 2.6, version variation can be caused by different reasons, such as requirement changes, non-uniform data, or custom fields in entities. We have used the *required* validator to specify which fields are part of a particular entity version. However, Mongoose provides the *discriminator* mechanism to have collections of non-uniform data types. This mechanism would be more appropriate than the *require* option for non-uniform data. For instance, a *MovieTheater* collection could register two kinds of movie theaters in our *Movie* database: single screen or multiplexed theaters. The *name*, *city*, *country* fields would be common, but the *noOfRooms* field would be only part of multiplexed theaters. Figure 7.5 shows how to declare a discriminator for an entity, *MovieTheater* in the example.

The schemas generated from this declaration would be the following:

```
var options = {discriminatorKey: 'kind'};

var movieTheaterSchema = new mongoose.Schema(
    {name: String, , city: String, country: String},
    options);
var MovieTheater1 = mongoose.model('MovieTheater1',
                                   theaterSchema);

var MovieTheater2 = MovieTheater1.discriminator(
    'MovieTheater2',
    new mongoose.Schema({noOfRooms: Number}, options));
```

Mongoose provides *update()* helper methods, but they do not apply validators, so the code to perform updating must be written following three steps (find-update-save). We also automate the generation of this code. For instance, in Figure 7.5 we show the code generated for updating the *genre* field of the *Movie* schema:

```
function update_genre(query, aGenre) {
   Movie.findOne (
      query,
      function (err, movie) {
         if (!err) {
            movie.genre = aGenre;
            movie.save(function (err, user) {
               console.log('Movie␣saved:␣', movie);
            });
         }
      }
   );
}
```

7.3 Generation of Data Validators

Validation is often needed when dealing with NoSQL databases. For instance, a developer would want to assure that all the objects retrieved and stored by a given application conform to a given entity version. When developing a new version of an application, for example, object validators (a.k.a. schema predicates) could be created so that the programmer can check each object that transfers to and from the database. Another scenario could be removing a given version of objects from the database, or migrating one version to another. Validators

allow checking, for example, that objects that are about to be inserted in the database comply with the required structure.

Using the *EntityDifferentiation* metamodel, via a M2T transformation using XTend [2], we built validation functions that check if a given JSON object is of the correct entity and entity version. Thus, the process generates validation functions for each entity version.

NoSQL databases differ from SQL databases in that they are *schema-on-read*, instead of SQL, which are *schema-on-write*. That is, the correctness of an given object with respect to a schema is checked once the object is read from the database. We can also mimic the behavior of SQL systems and we can check beforehand that a given object to be written into the database comply with the schema.

Thus, we have two different perspectives for validation:

- Validation before writing to the database, and
- Validation when reading from the database.

These two processes are subtly different in our case. When reading from the database, the objects are known to be of a given entity. The only uncertainty is what specific entity version do they belong to. Thus, in this case, the common properties of all the versions of an entity (those described by the *commonProps* attribute of the *EntityDiffSpec* class) do not have to be tested, as they will be present in all the obtained objects of a given entity. Only the properties described by *propertySpecs* attribute of the *EntityVersionProp* class have to be checked. This case will be shown in the next section (Section 7.4).

When writing into the database, the object produced by the application can have any set of arbitrary properties. Thus, the check in this case has to be complete, including the common and specific properties of the given entity version that the application wants to store.

In the rest of this Section we will study how validator functions are generated for each entity version. We will generate JavaScript code to perform the validation, as most of the databases considered use Javascript extensively either from the client or to write MapReduce processes. A Javascript client program could use these validator functions to assure that the objects that it writes to the database are correct with respect to the entity version they want to generate. Of course, other languages could be exercised by the M2T transformation.

The code generation can be described as follows:

1. For each discovered Entity, consider all of its Entity Versions.
2. For each Entity Version, generate *two* validator functions. One "minimal", and one "exact" function.
3. The "minimal" function refers to the minimum properties an object must possess to belong to an entity version. This could be considered as a form of *duck typing*. In this case, if the object has *more* fields than the entity version, it will be allowed. This is because sometimes the application would want to add additional attributes, while keeping compatibility by maintaining a set of known previous attributes.
   (a) To generate the "minimal" function, for each of the attributes of the entity version (common and specific) we generate a property check. Property checks, depending on the *needsTypeCheck* attribute of *PropertySpec*, have two forms:
      i. If *needsTypeCheck* is true, a test that checks whether this object has the property name *and* of the corresponding type is emitted. Type checks are tailored depending on the actual type, so checks for strings, integers, arrays, and aggregate objects are produced accordingly. In the case of aggregate objects, the corresponding validation function call is emitted for the aggregated type.
      ii. If *needsTypeCheck* is false, only a test for the existence of the property name within the object is emitted by the M2T transformation.
4. The "exact" function is very similar to the minimal function seen above. The main difference is that, apart from considering common and specific properties, also "property absence tests" are generated for the properties listed in the *notProps* attribute of *EntityVersionProp*. These absence tests just check that the object do not have the properties listed by *notProps*. This has to be done to treat the case in which an entity version's properties are a subset of the properties of another entity version. If the first check is performed, then the object would be erroneously considered to be of the first entity version, when it may be of the second entity version if it has the correct attributes.

Below we show some generated code for the running example database. Recall for example that the *MovieTheater* entity had two entity versions, one of them (*MovieTheater_2*) having the attribute *noOfRooms*. The code generated for *MovieTheater_2* is shown in Figure 7.6.

```
Movietheater_2: {
name: "Movietheater_2",
isOfExactType: function (obj)
{
    var b = true;
    b = b && ("type" in obj)
          && (obj.type.match(/Movietheater/i) ? true : false);
    b = b && ("_id" in obj);
    b = b && ("name" in obj);
    b = b && ("city" in obj);
    b = b && ("country" in obj);
    b = b && ("noOfRooms" in obj);
    return b;
},
isOfType: function (obj)
{
    var b = true;
    b = b && ("type" in obj)
          && (obj.type.match(/Movietheater/i) ? true : false);
    b = b && ("_id" in obj);
    b = b && ("name" in obj);
    b = b && ("city" in obj);
    b = b && ("country" in obj);
    b = b && ("noOfRooms" in obj);
    return b;
}
},
```

Figure 7.6: Validation Code for *Movietheater_2*.

The "minimal" function is in this case `isOfType`, and the "exact" one is `isOfExactType`. Both versions in this case are equal because this entity is the one that has more attributes, so checking all of them ensures an object is of the correct entity version. Note also how the *type* attribute is checked against the name of the Entity, and only non-type-check tests are performed.

The Figure 7.7, in turn, shows the code generated for *Movietheater_1*. The attribute *noOfRooms* is not checked in the minimal function, *but*, for the exact function, the absence

```
Movietheater_1: {
name: "Movietheater_1",
isOfExactType: function (obj)
{
    var b = true;
    b = b && ("type" in obj)
          && (obj.type.match(/Movietheater/i) ? true : false);
    b = b && ("_id" in obj);
    b = b && ("name" in obj);
    b = b && ("city" in obj);
    b = b && ("country" in obj);
    b = b && !("noOfRooms" in obj);
    return b;
},
isOfType: function (obj)
{
    var b = true;
    b = b && ("type" in obj)
          && (obj.type.match(/Movietheater/i) ? true : false);
    b = b && ("_id" in obj);
    b = b && ("name" in obj);
    b = b && ("city" in obj);
    b = b && ("country" in obj);
    return b;
}
},
```

Figure 7.7: Validation Code for *Movietheater_1*.

of the *noOfRooms* attribute must be checked to assure that the object does not effectively belongs to *Movietheater_2*.

Finally, Figure 7.8 shows an example of a type check for *Movie_3*. The *prizes* attribute changes from *Movie_3* and *Movie_1*, so in both has to be type checked. The figure also shows how the internal type of an aggregated object (in this case an array of objects *prizes*) is checked calling the previously generated `isOfExactType` of the corresponding type (*Prize_2*).

7.4 Entity Version Classification

As commented in the previous Section, validation on read is different from validation on write, as all the objects in the database, after the inference process, belong to one of the Entity Versions. Thus, for each object, the checks will be produced that tell its entity version.

These check functions can be used as the first step of, for example, migration processes

```
name: "Movie_3",
isOfExactType: function (obj)
{
    var b = true;
    b = b && ("type" in obj) && (obj.type.match(/Movie/i) ? true : false);
    b = b && ("director_id" in obj);
    b = b && ("title" in obj);
    b = b && ("_id" in obj);
    b = b && ("year" in obj);
    b = b && ("prizes" in obj) && (obj.prizes.constructor === Array) &&
        obj.prizes.every(function(e)
            { return (typeof e === 'object') && !(e.constructor === Array)
                && (
                mongoMovies3.Prize_2.isOfExactType(e)
                );
            })
    ;
    b = b && ("genre" in obj);
    b = b && !("genres" in obj);
    b = b && !("rating" in obj);
    b = b && !("running_time" in obj);
    b = b && !("writers" in obj);
    b = b && !("criticisms" in obj);
    return b;
},
```

Figure 7.8: Excerpt of the Validation Code for *Movie_3*.

that allow to change a given entity version into another version inside the database, or to obtain statistics from the data base.

As the set of checks is known for all the versions of a given entity, a *minimal set* of checks can be generated using algorithmic techniques. For that, a *Decision Tree* algorithm [75] was used to decide the minimum path that distinguished among entity versions. The different checks of the decision tree would be the presence or absence of attributes from the *Entity-Differentiation* model.

To store the information of the different decision trees for the entities, we built a *DecisionTree* metamodel. This metamodel captures the root for each entity (*DecisionTreeForEntity*), and a set of *DecisionTreeNode*s, which can be either *LeafNode*s, that identify a given entity version, or *IntermediateNode*s, that check a given property of the object (*checkedProperty*). The intermediate nodes have a *yesBranch* and a *noBranch* property, to form the tree, and correspond to the result of the check for the property performed in the object.

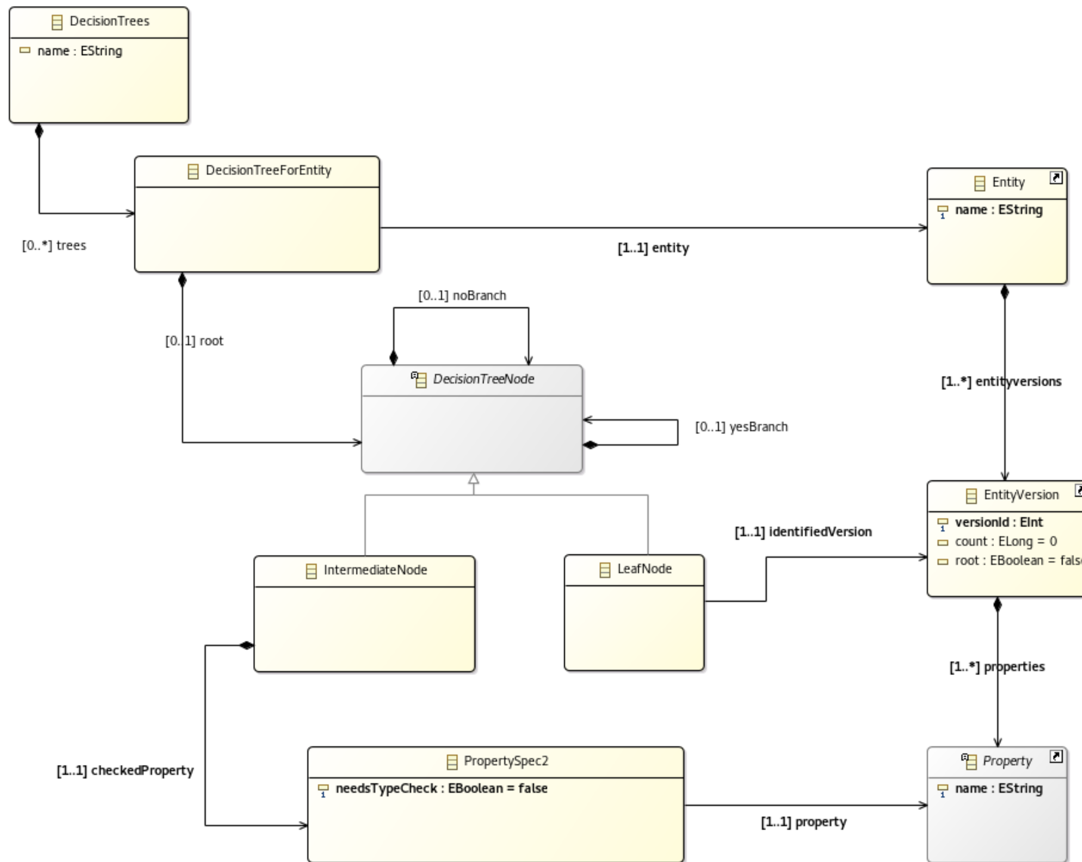

Figure 7.9: Decision Tree Metamodel.

To build *DecisionTree* models, a M2M process from the *EntityDifferentiation* models has been written in Java. Using all the properties of each Entity Version, a Weka decision tree algorithm has been implemented. For all the properties of all the entity versions, Weka instances were created, and the decision tree algorithm executed. The tree, then, was interpreted to finish the M2M process and generate the final *DecisionTree* model.

An example logical tree generated for the *Movie* entity can be seen in Figure 7.10. Only a small set of attributes is checked to determine that an object belongs to an entity version. This is because the decision tree algorithm selects the optimal path that differentiates the different instances (entity versions, in this case). As can be seen in the Figure, all the entity versions have the same weight (1.0), but in a future work we will give weights according to the number of objects of that entity version present in the database (this is obtained as part

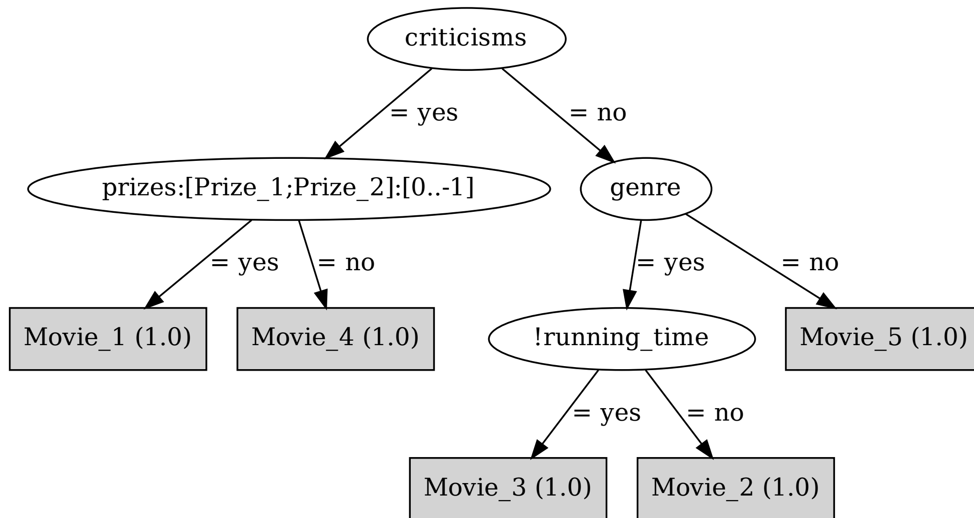

Figure 7.10: Decision Tree for the *Movie* Entity.

of the MapReduce process), to prioritize the checks generated.

Note that different checks are generated, depending on the *needsTypeCheck* flag (see for example *prices* in the first left branch of the tree). Also, "absence tests" are generated, specified with a "!" prepending the name (see *!running_time* in the figure).

The generated code for the *Movie* entity is shown in Figure 7.11 (`entityVersionForObject` function). The nested `if` constructs mimic those of the decision tree branches. Note how for *prices* a full type check is emitted. Also, thanks to the properties of the decision tree, we can exchange the "yes" and "no" branches of the tree shown in Figure 7.10 for the "absence test" branches, like for example *!running_time*, and generate just normal existence tests. We used here again Javascript, as these check functions can be used, for example, inside MapReduce scripts for databases to perform the selection of objects to process based on their entity version.

Finally, using the same information we can also generate specific check functions for specific entity versions. These tests will have the minimum set of checks needed to tell if a given object belongs to a given entity version. Figure 7.12 shows the code generated for some entity versions. As can be seen, the branches of the tree are followed to generate the tests.

```
name: "Movie",
entityVersionForObject: function (obj)
{
    if (("criticisms" in obj))
    {
        if (("prizes" in obj) && (obj.prizes.constructor === Array) &&
            obj.prizes.every(function(e)
                { return (typeof e === 'object')
                    && !(e.constructor === Array)
                    && (
                    mongoMovies3.Prize_1.isOfExactType(e) ||
                    mongoMovies3.Prize_2.isOfExactType(e)
                    );
                })
        )
        {
            return "Movie_1";
        } else {
            return "Movie_4";
        }
    } else {
        if (("genre" in obj))
        {
            if (("running_time" in obj))
            {
                return "Movie_2";
            } else {
                return "Movie_3";
            }
        } else {
            return "Movie_5";
        }
    }
}
```

Figure 7.11: Decision Tree Code for the *Movie* Entity.

```
checkEV_Movie_2: function (obj)
{
    if (("criticisms" in obj))
        return false;
    if ((!("genre" in obj)))
        return false;
    if ((!("running_time" in obj)))
        return false;

    return true;
},
checkEV_Movie_3: function (obj)
{
    if (("criticisms" in obj))
        return false;
    if ((!("genre" in obj)))
        return false;
    if (("running_time" in obj))
        return false;

    return true;
},
checkEV_Movie_5: function (obj)
{
    if (("criticisms" in obj))
        return false;
    if (("genre" in obj))
        return false;

    return true;
}
```

Figure 7.12: Check Functions for *Movie* versions 2, 3, and 5.

It would be great to have a undo function in our lives

Anonymous

8
Conclusions and Future Work

Interest in NoSQL systems is continuously growing and the database reports predict that they will be widely adopted by a large number of companies in the near future. Actually, a polyglot persistence is the foreseen scenario, in which companies will use both relational databases and different kinds of NoSQL stores [1, 109].

Database schemas are a valuable asset, traditionally known to bring several benefits: (i) they help in understanding how data is stored and organized in a database; (ii) they allow to statically detect the errors in data access made in programs; and (iii) they provide knowledge required by tools that provide functionality such as facilitating homogeneous query languages over data or integrating heterogeneous databases.

While the concept of database schema plays a central role in relational database systems, most NoSQL systems are schemaless, that is, they do not require having to formally define an schema. NoSQL schemaless databases also have schemas but they are implicit into stored data and application code. Not having to define schemas is necessary because this feature offers the flexibility that modern applications demands due to that the data structure frequently changes. Certainly, the absence of an explicit schema is a very attractive characteristic for many NoSQL developers. However, it is becoming increasingly evident that the schemaless nature in NoSQL database should coexist with a knowledge of the schema by

tools that help to developers. The schemas would be discovered by means of reverse engineering processes [1]. The Dataversity report [1] evidenced the necessity of building such tools to support the development of NoSQL applications. Tools with a functionality similar to those offered for relational databases are required, specially tools for (i) generating code, (ii) model visualization, and (iii) metadata management.

Therefore, the inferred schemas of NoSQL databases are useful to build a number of tools intended to help developers that make use of NoSQL databases. They may mitigate the problems due to the lack of an explicit schema. For instance, reports, diagrams, validators, and version migration scripts could be automatically generated from the NoSQL Schema models. Moreover, there are tools that require knowledge of the schema in order to provide certain functionality, e.g. SQL query engines or integrating heterogeneous databases.

Building these tools poses some challenges that demand a great effort of industry and academia in the emerging *NoSQL Data Engineering* area. When the work of this thesis started at the beginning of 2014, few research works in this area had been published. At the time of writing this thesis, the research effort in that area have been still very limited, and it has been mainly focused on the inference of schemas from data stored in document stores, as discussed in Chapter 3. Regarding to database tools, some existing modeling tools for designig relational databases are being extended to provide NoSQL inferred schema visualization [47, 37, 48]. Moreover, NoSQL systems are offering tools for viewing, analyzing, and querying stored data, as Compass for MongoDB [33]. Some kind of data analysis is also performed in [72] where outliers are identified.

Next, we will discuss the level of achievement of the goals of this thesis that were presented in Chapter 1.

8.1 Discussion

In Chapter 1, five objectives were defined: (1) to design and implement a schema inference process for aggregate-based NoSQL systems; (2) to propose of a notion of NoSQL data schema; (3) to design diagrams for representing NoSQL schemas and to implement tools that support its visualization; (4) Code generation for ODM mappers; and (5) Generation of data validators.

Next, we shall discuss to what extension the goals have been achieved.

8.1.1 Goal 1. Design and Implementation of a Schema Inference Process

We have defined a schema discovering approach that has been implemented as a model-driven reverse engineering process as described in Chapter 5. The schema inferred is represented as a model that conforms to the NoSQL_Schema metamodel which is NoSQL system-independent. The main differences of our inference strategy with respect to other proposed approaches are the following: (i) to extract the versions of each entity; (ii) to discover all the relationships among the entity versions extracted: aggregation and references; (iii) consider the scalability and performance of the inference algorithm by applying a MapReduce operation to directly access to the database and obtaining the minimum set of JSON objects needed to apply the inference process. Our interest is not to obtain a succinct, approximate or skeleton schema. Instead, our idea is to record all the entity versions and relationships between them. This decision is motivated by the fact that our approach is targeted to business applications in which the maximum number of versions of an entity will not be very high. This scenario is different of that considered in [119], that supposes that several tens of thousand of version can exist for an entity.

The approach has been validated with a real case study, in particular we have created a MongoDB database from the open data Stackoverflow dataset. This validation showed that the inference algorithm scaled well with respect to the number of objects.

8.1.2 Goal 2. Proposal of a notion of NoSQL data schema

As far as we know, our work is the first approach that manages versioned schemas as shown in Table 3.1. Other approaches do not take into account versions of entities [23] or either they obtain the union schema [72] or an approximate schema [119]. Here, we have defined the notion of Versioned Schema, and the following kinds of versioned schemas: *Version Schema* (Root or Aggregate) that only includes the entity versions related to an entity version (and entities if there are references); *Entity Schema* (Root or Aggregate) that only includes entities related to an entity; *Database Schema* that includes all the entity versions of a database; and *Entity Database Schema* that includes the union schemas of all the entities of the database. Therefore, we have identified a set of schemas that can be considered in deal-

ing with NoSQL systems, which can be defined to three levels: entity, entity version, and database. These schemas have been defined in detail in Section 4.1.

The schemas are valid for document-based NoSQL databases, but it can be applied to wide column stores too, as stated in the future work. Adaptation to graph-based databases is also possible.

8.1.3 Goal 3. Design Diagrams for NoSQL Schemas and Implement Tools that Support its Visualization

Developers of NoSQL database applications need to understand the implicit database schema. In fact, they must keep in mind this schema when they write or maintain code. The visualization of the schema in form of diagrams would be very helpful for these developers in the same way as the E/R schemas have been used for developers of relational database applications. We have defined visual representations for each kind of schema defined in this thesis. In particular, we have used UML class diagrams to represent root version schemas, entity union schemas, and entity database schemas. Several benefits are gained by representing NoSQL schemas in form of these diagrams: both understanding about them and its communication are facilitated, and a documentation separated from the code is obtained.

We have developed two MDE solutions to visualize the kinds of diagrams defined. First, we transform the inferred schema model generated into a Ecore metamodel with the aim of visualizing it by means of an Ecore metamodel editor. We have used for this the editor integrated into Eclipse/EMF. This solution have illustrated the benefits of representing models and metamodels uniformly. After this proof of concept, we transform the schema model inferred into PlantUML code for a visualization of UML class diagram. Our work has served to define a specific notation for NoSQL schemas [29].

Recently, some companies that offer data modeling tools have extended theirs tools to provide some kind of visualization of schemas for document stores (normally MongoDB) [47, 37]. However, these tools do not cope well with the variability of the schemaless data: they either do not support variation in the structure of the objects of a given type, or they overgeneralize the schema to embrace all the possible variations. In our proposal, we define schemas that take into account the existing versions of each type: the versioned schemas have the unique characteristic of completely defining the structure of the data, also

showing the high-level relationships, such as aggregation and reference.

8.1.4 Goals 4 and 5. Code Generation for ODM Mappers and Data Validators

With the aim of illustrating some possible application of schemas inferred in addition to its visualization, we have tackled the development of two code generation utilities: code for ODM mappers and code for data validation.

We have designed and implememented the generation of the schema specification code for existing ODM mappers.In the case of Mongoose, we have also been able to generate artefacts for different functionality provided by this mapper, such as validators, discriminators, and reference management. The MDE solution devised has shown the usefulness of defining intermediate metamodels in a model transformation chain. We have defined the *EntityDifferentiation* metamodel, which reorganizes the information included in a schema model in a more appropriate form to generate the schemas of each version. This requires to distinguish between common and specific properties for each version.

Data validation is needed to assure that all the objects retrieved and stored by a given application conform to a given entity version. We have developed an MDE solution that generates validators to be applied when data are stored into the database. This solution has been implemented as a two-step model transformation chain. Here, a metamodel has also been defined to have an intermediate representation that facilitates the generation of validators. The intermediate models are obtained via a model-to-model transformation from the NoSQL_Schema models, and then a model-to-text transformation generates the validator functions that check the entity version of the data to be stored.

8.1.5 Goal 6. Definition of a Data Classifier

To determine the entity version an object from the database belongs to, a clustering algorithm has been designed, based on a decision tree algorithm. This algorithm assures that minimal checks are performed in order to discriminate between the different entity versions of an object belonging to a given entity. This classification allows filtering and applying transformation only to those objects that belong to a given entity version.

8.1.6 Advantages of using MDE

It is worth noting that a general objective of this thesis was to show the benefits of using MDE in the emerging area of NoSQL Data Engineering. We have created MDE solutions both to implement the schema inference process and the database utilities developed to illustrate possible applications of the inferred schema models. These solutions have shown some of the main advantages that MDE offers as:

- **Represent information at a high level of abstraction**. Metamodeling is a more expressive formalism than XML and JSON when representing the information involved in a data reverse engineering process. Rather than using proprietary formats, models allow the information to be uniformly represented, which favors software quality, e.g. interoperability, extensibility or reuse. The existence of widely adopted metamodeling languages (e.g. Ecore) strengthens the benefit of metamodels with regard to proprietary formats. We have defined metamodels (i) to represent NoSQL schemas, and (ii) to obtain intermediate representations that reorganize the information included in schema models in an adequate format to generate code, and (iii) to represent the decision tree defined for classifying data into versions.

- **Everything is a model**. As noted in [15], the fact that metamodels and meta-metamodels are also models is a strength of MDE. In this thesis, we have been able to appreciate the unification power of models in the first strategy of visualization of schemas based on a transformation that convert a schema model into an Ecore metamodel.

- **Automation**. We have created model transformation chains to implement all the tool developed in this thesis. We have been able to automatically generate code of different platforms: PlantUML DSL, Javascript code of Mongoose schemas, Java validators. Here we have not measured the gain of productivity as we did in previous works [62, 63, 106] because this was not the focus of our work.

- **Tooling for building DSLs**. Several metamodel-based DSL definition tools there are available to build DSL. Xtext and Sirius are two of the more widely used tools in the Eclipse platform, Xtext for textual DSLs and Sirius for graphical DSLs.We have been used Xtext to create the *Parameter DSL* aimed to express the parameter model

required to generate code for the Mongoose mapper. Sirius has been used in [29] to define a graphical notation aimed to represent diagrams of NoSQL schemas. These tools allow to save a great effort and time in developing DSLs.

- **Platform independence**. Independence of source and target technologies can be achieved the use of models. A pivot model can be used to achieve this independence. In our case, the schema metamodel is independent of a particular aggregate-based NoSQL system, and he plays the role the unified or pivot model.

8.2 Contributions

The main contributions of our work have been the following. To our knowledge, we have defined the first approach that infers database schemas from NoSQL databases discovering all the versions of the inferred entities and their relationships. Other novelty aspect of this approach is to address the scalability. In addition, the schema metamodel contributes to the proposals of unified models for NoSQL databases. The first version of the approach was presented in April, 2015 in the XVIII Iberoamerican Conference on Software Engineering (CIbSE'2015) [82] and the second version in the 34th International Conference on Conceptual Modeling (ER 2015) [102].

Our analysis of the notion of schema for aggregate-based NoSQL databases is another contribution of this thesis. The set of proposed schemas could serve to guide further research work in this domain. As far as we know, we have presented the first proposal to visualize NoSQL versioned schemas, both the usage of UML class diagram and the definition of a specific notation. The first utility to visualize schemas was already outlined in the article presented in ER'2015. Now, we are preparing and article that describes all our work on visualization of NoSQL schemas, which will be submitted to the 36th International Conference on Conceptual Modeling (ER'2017) that will be held in November, 2017 in Valencia (Spain) [59]. A previous version of this work will be presented in the Spanish Conference on Software Engineering and Databases (JISBD'2017) [60] that will be held in July, 2017 in La Laguna (Tenerife, Spain).

To our knowledge, we have designed and implemented the first utilities for the generation of code for ODM mappers. In particular, we have generated code for the Mongoose

mapper, but the MDE solution defined is applicable to any ODM mapper. This approach has been presented in the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017) [104]. We have also shown novelty approaches to generate data validators and classify data into entity versions. The development of these tools has also contributed to show how MDE techniques can be very useful in NoSQL Data Engineering area. While the application of MDE in the Data Engineering field has been very limited to date, we believe that experiences of usage as the here described can help to understand the benefits of its usage and motivate to NoSQL tool builders to take advantage of MDE technology. We introduced the "Model-driven NoSQL Data Engineering" term in a work presented to the Spanish Conference on Software Engineering and Databases (JISBD'2015) [103] in which we pointed out the emergence of the NoSQL Data Engineering research area and contemplated the application of MDE in that area. We are now preparing a longer paper that describes the current version our schema discovering strategy and the two mentioned code generation utilities are used to illustrate the possible applications of the inferred schemas. This paper will be submitted to the "Information and Software Technology" journal.

The study of the state of the srt made for this thesis (Chapter 3) is another contribution of our work. We have contrasted the NoSQL schema discovering approaches proposed, unified metamodels, and tooling that are available at this moment. This analysis has discussed the more significant academic and industrial efforts in the NoSQL data engineering area. We have also identified a set of criteria to compare the different schema inference approaches. To our knowledge, no review as exhaustive as the one presented here has been published to date.

Cites At the time of writing this thesis, our work has received only 6 cites (we have consulted Google Scholar) and they only reference the article presented in ER'2015 [102]. The proceedings of the article that describes the generation of code for ODM mappers has been published a few weeks ago, so this work has not been cited yet.

In [65] is noted that our approach "could be applied to analyze JSON documents, however, it is specially tailored to NoSQL databases and do not provide assistance to integrate Web APIs". In [36], it is compared with a proposal to generate graph databases from UM-

L/OCL conceptual schemas and it is noted that "they do not aim to provide support for a full-fledged application nor consider the addition of constraints on the reversed schema". Our work is also considered in a recent study on data modeling in the NoSQL world [12], in which it is claimed that our approach "supports the idea that, even in the NoSQL context, a model-based description of the organization of data is very useful during the entire life-cycle of a data set". In a work on the definition of a standarization model for integrating heterogeneous databases is indicated that "our idea of reverse engineering the database to obtain the schemas in the original models can also be useful in the case of the graph model" [10]. [54] considers that our approach "introduces schema management on top of schemaless document systems". Finally, our work is discussed together with the approaches [72] and [119] in a paper that presents a proposal for finding multidimensional structures in document stores aimed at enabling OLAP querying in the context of self-service Business Intelligence (BI) [30]; the authors commented that "these works focus on the structural variety of documents within the same collection caused by their schemaless nature and by the evolution of data".

8.3 Future Work

The research we have presented in this thesis has allowed the Modelum Group and the Catedra SAES-UMU to start a research line in Model-Driven NoSQL Data Engineering, in which they will collaborate in the following years. Several interesting directions can be taken to continue our research activity in this area. They are here organized according to the main objectives of this thesis. In addition, we point out other directions that present a not so direct relationship with the topics here addressed. Note that some future works outlined differ in the scope.

Schema inference process We will adapt our reverse-engineering process to be applied to column-family stores. This involves adapting the inference process, as some of these databases do not have proper data types, and only store data blobs, for example HBase [57]. Moreover, we will validate our inference process with more open datasets such as DBPedia or IMDB. A more fine type system for the data inference is planned. With this, more detailed types can be output for schemas, supporting attributes that only are of a set possible

values (similar to enums in some programming languages), or that only hold several ranges of values, or union types to better support variability. Generating type specifications similar to Datalog is also planned.

Schema visualization When several tens or hundreds of schemas are inferred, the visualization of all the schemas is not helpful, but queries or browsing mechanisms are needed to understand the database schemas. We are extending the schema editor developed with Sirius with such capabilities. Also, joining mechanisms such as optional fields will allow to reduce the number of versions.

Data visualization In this thesis we have explored how data analysis techniques can be applied to generate data visualizations that take into account the schema *and* version of the objects in the database, allowing to visually identify the quantities of objects of each type and version. If a data base has evolved over time, it would be interesting to show which data belong to each version. After this initial effort, future directions include allowing more statistics to be done in the data and to be shown related to their entity version.

Code generation The MDE techniques and metamodels used in thesis has proven very useful for generating different artefacts and utilities for NoSQL databases. More utilities can be generated, however. We plan, among other research works, to consider more mappers for MongoDB and for other data bases, and to build a generative architecture to automate the building of Mongoose-based MEAN applications for existing databases.

Database evolution When the database evolves, data that corresponds to new entity versions can be stored. Then, it could be required to migrate data from old versions to the new version. Generating object version transformers could then be interesting. A developer can describe, by means of a specialized DSL, the necessary steps to convert one version of an object to another version. These could be used in at least two ways:

- A new application that uses the stored old data may require that all the recovered objects comply with the new version. A version transformer could be generated that removes the unneeded fields, and gives values to new, non-existing fields. This would

guarantee that the application would always use object with the correct (new) version, giving all the process more robustness.

- Batch database migration. MapReduce jobs could be generated to transform old version objects into new versions. This is possible given the precise version information stored in the schema.

Design of NoSQL databases The design of NoSQL stores is mainly influenced by the queries to be issued on the database. We are investigating how the the graphical notation defined to visualize schemas could be converted into a DSL aimed to design NoSQL database. This DSL would allow to express information on queries that could be used to generate the database schema. Moreover, we are investigating how execution plans of SQL queries could be used to migrate relational schemas to NoSQL schemas.

Relational database to Polyglot persistence migration As indicated in the report [1] "Polyglot persistence is the new normal: As NoSQL database architectures are specialized, people need to use multiple systems in their enterprises, leading to polyglot persistence." Considering the experience of the ModelUM group in data and code reengineering, we are exploring how tackle the challenges of the migration of legacy applications to new systems based on polyglot persistence.

8.4 Publications

The research activity of this thesis has produced various contributions that have been presented and discussed on several peer-review forums. The articles in which the research from this thesis has been published are presented below.

- Sevilla Ruiz, D., Feliciano Morales, S. and García-Molina, J.: *An MDE Approach to Generate Schemas for Object-document Mappers*. In Luis Ferreira, Slimane Hammoudi and Bran Selic (eds), Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), pages 220-228, SCITEPRESS, 2017 (Candidate to best paper award, not published acceptance rate).

- Sevilla Ruiz D., Feliciano Morales, S., García-Molina J.: *Inferring Versioned Schemas from NoSQL Databases and Its Applications*. In: Johannesson P. et al. (eds), Proceedings of Conceptual Modeling, 34th International Conference (ER 2015), pages 467–480. Lecture Notes in Computer Science, vol. 9381, Springer, 2015. (acceptance rate of 20%).

- Feliciano Morales, S. and García-Molina, J. and Sevilla Ruiz, D.: *Inferencia del esquema en bases de datos NoSQL a través de un enfoque MDE*. In Araujo, J. et al. (eds), Proceedings of 18th IberoAmerican Conference on Software Engineering (CIbSE 2015), pages 11–25, Curran Associates. April 22-24, 2015. (acceptance rate of 22,5%).

- Hernandez, A. Sevilla Ruiz D., Feliciano Morales S., García-Molina J.: *A Model-Driven approach to visualize NoSQL schemas*. Submitted to the 36th International Conference on Conceptual Modeling (ER 2017).

- Hernandez, A. Sevilla Ruiz D., Feliciano Morales S., García-Molina J.: *Visualización de Esquemas en Bases de Datos NoSQL*. In Ruiz González, F (Eds.), Actas de las XXII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2017). La Laguna, Tenerife, julio, 2017.

- Sevilla Ruiz D., Feliciano Morales S., García-Molina J.: *Model-Driven NoSQL Data Engineering*. In Canós, J. H. y González-Harbour, M. (Eds.), Actas de las XX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2015). Santander, septiembre, 2015 (handle: 11705/JISBD/2015/021)

8.5 Software Utilities Developed

The tools implemented in this thesis can be downloaded from `https://github.com/catedrasaes-umu/NoSQLDataEngineering`.

References

[1] (2015). Insights into NoSQL Modeling: A dataversity Report.

[2] (2017). Xtend website official. http://www.eclipse.org/xtend/. Last access: March 2017.

[3] (Visited March 2017a). JavaScript JSON. http://www.w3schools.com/js/js_json.asp.

[4] (Visited March 2017b). Json schema. http://json-schema.org/.

[5] (Visited March 2017). Main Web Page of CouchDB. http://couchdb.apache.org/.

[6] (Visited March 2017). Metaedit website official. http://www.metacase.com/mep/.

[7] (Visited March 2017). Sirius website official. https://eclipse.org/sirius/.

[8] Abiteboul, S. (1996). *Querying Semi-Structured Data*. Technical Report 1996–19, Stanford InfoLab.

[9] Abiteboul, S., Buneman, P., & Suciu, D. (1999). *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann.

[10] Aggarwal, D. & Davis, K. C. (2016). Employing graph databases as a standardization model towards addressing heterogeneity. In *17th IEEE International Conference on Information Reuse and Integration, IRI 2016, Pittsburgh, PA, USA, July 28-30, 2016* (pp. 198–207).

[11] Atzeni, P., Bugiotti, F., & Rossi, L. (2012). Uniform access to non-relational database systems: The SOS platform. In *Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings* (pp. 160–174).

[12] Atzeni, P., Bugiotti, F., & Rossi, L. (2014). Uniform access to nosql systems. *Information Systems*, 43, 117 – 133.

[13] Bertino, E. & Martino, L. (1993). *Object-Oriented Database Systems*. Addison-Wesley.

[14] Bex, G. J., Neven, F., Schwentick, T., & Vansummeren, S. (2010). Inference of concise regular expressions and dtds. *ACM Trans. Database Syst.*, 35(2), 11:1–11:47.

[15] Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2), 171–188.

[16] Brambilla, M., Cabot, J., & Wimmer, M. (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers.

[17] Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014). MoDisco: A Model Driven Reverse Engineering Framework. *Information & Software Technoligy*, 56(8), 1012–1032.

[18] Brunelière, H., Cabot, J., Izquierdo, J. L. C., Arrieta, L. O., Strauß, O., & Wimmer, M. (2015). Software modernization revisited: Challenges and prospects. *IEEE Computer*, 48(8), 76–80.

[19] Bugiotti, F., Cabibbo, L., Atzeni, P., & Torlone, R. (2014). Database design for nosql systems. In *Conceptual Modeling - 33rd International Conference, ER 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings* (pp. 223–231).

[20] Buneman, P. (1997). Semistructured data. In *Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (pp. 117–121).: ACM.

[21] Cassandra (Visited March 2017). Apache cassandra. http://cassandra.apache.org/.

[22] Castrejón, J., Vargas-Solar, G., Collet, C., & Lozano, R. (2013). Exschema: Discovering and maintaining schemas from polyglot persistence applications. Available on ResearchGate (2016, extended version of the paper published in ICSM'2013.

[23] Castrejón, J. C., Vargas-Solar, G., Collet, C., & Lozano, R. (2013). ExSchema: Discovering and Maintaining Schemas from Polyglot Persistence Applications. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013* (pp. 496–499).

[24] Cattell, R. G. G. (1994). *Object Data Management: Object-Oriented and Extended Relational Database Systems (Revised Edition).* Addison-Wesley.

[25] Chen, P. P. (1976a). Ansi/x3/sparc study group on data base management systems. interim report. *ACM SIGMOD bulletin*, 7(2).

[26] Chen, P. P. (1976b). The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1), 9–36.

[27] Chi, Y., Yang, Y., & Muntz, R. R. (2005). Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowl. Inf. Syst.*, 8(2), 203–234.

[28] Chikofsky, E. J. & Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1), 13–17.

[29] Chillón, A. H. (2016). Visualización de Esquemas y Datos en Bases de Datos NoSQL. Master's thesis, Facultad de Informática. Universidad de Murcia. Spain.

[30] Chouder, M. L., Rizzi, S., & Chalal, R. (2017). Enabling self-service BI on document stores. In *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017*.

[31] Colazzo, D., Ghelli, G., & Sartiani, C. (2012). Typing massive json datasets. In *International Workshop on Cross-model Language Design and Implementation*, volume 541 (pp. 12–15).

[32] Communications, L. (2000). Whatever Happened to Object-Oriented Databases. http://leavcom.com/articles/db_08_00.htm.

[33] Compass (2017). Mongodb Compass Web Page. https://www.mongodb.com/products/compass. Accessed: March 2017.

[34] Cuadrado, J. S., Molina, J. G., & Menárguez, M. (2006). RubyTL: A practical, extensible transformation language. In *2nd European Conference on Model-Driven Architecture*, volume 4066 of *LNCS* (pp. 158–172).: Springer.

[35] Czarnecki, K. & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 621–646.

[36] Daniel, G., Sunyé, G., & Cabot, J. (2016). Umltographdb: Mapping conceptual schemas to graph databases. In *Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings* (pp. 430–444).

[37] DbSchema (Visited March 2017). Dbschema web page. http://www.dbschema.com/index_es.html.

[38] Dean, J. & Ghemawat, S. (2004). Mapreduce: Simplified Data Processing on Large Clusters. *OSDI*.

[39] Dean, J. & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1), 107–113.

[40] Discoverer, J. (Visited: March 2017). JSON Discoverer. http://som-research.uoc.edu/tools/jsonDiscoverer/#/.

[41] Doctrine (Visited March 2017). Doctrine project. http://www.doctrine-project.org/.

[42] Drill, A. (Visited: March 2017). Apache Drill. https://drill.apache.org/.

[43] Eclipse (2011). MOFScript v1.4. http://eclipse.org/gmt/mofscript/.

[44] Eclipse (2012). Acceleo. http://www.eclipse.org/acceleo/.

[45] emf2gv (Visited: March 2017). EMF To GraphViz. http://emftools.tuxfamily.org/wiki/doku.php?id=emf2gv:start.

[46] EMFatic (Visited: March 2017). Emfatic: A textual syntax for EMF/Ecore (meta-)models. https://www.eclipse.org/emfatic/.

[47] ER-Studio (2017). ER-Studio Web Page. https://www.idera.com/er-studio-enterprise-data-modeling-and-architecture-tools. Accessed: March 2017.

[48] ERWin (2017a). CA ERwin Web Page. http://erwin.com/products/data-modeler. Accessed: March 2017.

[49] ERWin (Visited March 2017b). EMFText Web Page. http://www.emftext.org/index.php/EMFText.

[50] Evans, E. (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[51] Fleurey, F., Breton, E., Baudry, B., Nicolas, A., & Jézéquel, J.-M. (2007). Model-driven engineering for software migration in a large industrial context. In *Proceedings of the MoDELS'07*, volume 4735 of *LNCS* (pp. 482–497).: Springer.

[52] Fowler, M. (2013). Schemaless Data Structures. http://martinfowler.com/articles/schemaless/.

[53] Gardner, T., Griffin, C., Koehler, J., & Hauser, R. (2003). A review of omg mof 2.0 query/views/transformations submissions and recommendations towards the final standard. In *MetaModelling for MDA Workshop*, volume 13 (pp.41).

[54] Gómez, P., Casallas, R., & Roncancio, C. (2016). Data schema does matter, even in NOSQL systems ! In *Actes du XXXIVème Congrès INFORSID, Grenoble, France, May 31 - June 3, 2016.* (pp. 207–208).

[55] GraphViz (Visited: March 2017). GraphViz Visualization Software. http://www.graphviz.org/.

[56] Hainaut, J.-L. (2006). The transformational approach to database engineering. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science* (pp. 95–143).: Springer.

[57] Hbase (Visited March 2017). Apache hbase. https://hbase.apache.org/.

[58] Hegewald, J., Naumann, F., & Weis, M. (2006). Xstruct: Efficient schema extraction from multiple and large XML documents. In *Proceedings of the 22nd International Conference on Data Engineering Workshops, ICDE 2006, 3-7 April 2006, Atlanta, GA, USA* (pp.81).

[59] Hernández, A., Ruiz, D. S., , Morales, S. F., & Molina, J. G. (2017a). A model-driven approach to visualize nosql schemas. In *Submitted to Conceptual Modeling - 36th International Conference, ER* Valencia, Spain.

[60] Hernández, A., Ruiz, D. S., Morales, S. F., & Molina, J. G. (2017b). Visualización de esquemas en bases de datos nosql. In *Spanish Conference on Software Engineering and Databases (JISBD'2017)* La Laguna, Tenerife, Spain.

[61] Holmes, S. (2013). *Mongoose for Application Development*. PACKT Publishing.

[62] Hoyos, J. R., Molina, J. G., & Botía, J. A. (2013). A domain-specific language for context modeling in context-aware systems. *Journal of Systems and Software*, 86(11), 2890–2905.

[63] Hoyos, J. R., Molina, J. J. G., Botía, J. A., & Preuveneers, D. (2016). A model-driven approach for quality of context in pervasive systems. *Computers & Electrical Engineering*, 55, 39–58.

[64] Izquierdo, J. L. C. & Cabot, J. (2013). Discovering implicit schemas in JSON data. In *Web Engineering - 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings* (pp. 68–83).

[65] Izquierdo, J. L. C. & Cabot, J. (2016). Jsondiscoverer: Visualizing the schema lurking behind JSON documents. *Knowl.-Based Syst.*, 103, 52–55.

[66] Janga, P. & Davis, K. C. (2013). *Schema Extraction and Integration of Heterogeneous XML Document Collections*, (pp. 176–187). Springer Berlin Heidelberg: Berlin, Heidelberg.

[67] Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2), 31–39.

[68] K., P., T., T., MA., R., & S., C. (2008). A design science research methodology for information systems research. In *Journal of Management Information Systems*, volume 24 (pp. 45–77).

[69] Karwin, B. (2015). Are Object-Relational Databases the Future? https://www.quora.com/Are-Object-relational-databases-the-future.

[70] Kelly, S. & Tolvanen, J. (2008). *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley.

[71] Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained*. Addison-Wesley.

[72] Klettke, M., Scherzinger, S., & Störl, U. (2015). Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *BTW*, volume 2105 (pp. 425–444).

[73] Kolovos, D., Paige, R., & Polack, F. (2008). The epsilon transformation language. In A. Vallecillo, J. Gray, & A. Pierantonio (Eds.), *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science* (pp. 46–60). Springer Berlin Heidelberg.

[74] Kurtev, I., Bézivin, J., & Aksit, M. (2002). Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*.

[75] Machine Learning Group at the University of Waikato (Visited: March 2017). Weka. http://www.cs.waikato.ac.nz/~ml/weka/.

[76] Mandango (Visited March 2017). Mandango website. https://mandango.org/.

[77] Meyer, B. (1997). *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.

[78] Min, J.-K., Ahn, J.-Y., & Chung, C.-W. (2003). Efficient extraction of schemas for xml documents. *Information Processing Letters*, 85(1), 7 – 12.

[79] Moh, C., Lim, E., & Ng, W. K. (2000). Dtd-miner: A tool for mining DTD from XML documents. In *Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000), Milpitas, California, USA, June 8-9, 2000* (pp. 144–151).

[80] MongoDB (2017). MongoDB Web Page. https://www.mongodb.com/. Accessed: March 2017.

[81] Mongoose (2017). Mongoose Web Page. http://mongoosejs.com. Accessed: March 2017.

[82] Morales, S. F., Molina, J. G., & Sevilla, D. (2015). Inferencia del esquema en bases de datos NoSQL a través de un enfoque MDE. In *Proceedings of 18th IberoAmerican Conference on Software Engineering (CIbSE 2015)* (pp. 11–25).: Curran Associtès.

[83] Morphia (Visited March 2017). Morphia github repository. https://github.com/mongodb/morphia.

[84] MPS (Visited March 2017). Metaprogramming system website official. https://www.jetbrains.com/mps/.

[85] Neo4J (Visited March 2017). Neo4j website official. https://neo4j.com/.

[86] Nestorov, S., Abiteboul, S., & Motwani, R. (1998). Extracting schema from semistructured data. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.* (pp. 295–306).

[87] NoSQL-Market (2016). NoSQL Market. https://www.alliedmarketresearch.com/NoSQL-market. Accessed: November 2016.

[88] NoSQL Ranking (2016). NoSQL Ranking. http://db-engines.com/en/ranking. Accessed: November 2016.

[89] OMG (2003). *MDA Guide Version 1.0.1. Object Management Group (OMG).* http://www.omg.org/mda.

[90] OMG (2011). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1.

[91] OMG (2012). OMG Object Constraint Language (OCL), Version 2.3.1.

[92] OMG (2013). OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1.

[93] OMG (Last access: March 2017). Architecture-Driven Modernization. http://adm.omg.org/.

[94] OrientDB (Visited March 2017). Orientdb website official. http://orientdb.com/orientdb/.

[95] Pérez-Castillo, R., de Guzmán, I. G. R., Piattini, M., & Ebert, C. (2011). Reengineering technologies. *IEEE Software*, 28(6), 13–17.

[96] PlantUML (Visited: March 2017). PlantUML in a nutshell. http://plantuml.com/.

[97] Redis (Visited March 2017). Redis. https://redis.io/.

[98] Redmond, E. & Wilson, J. R. (2012). *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf.

[99] Rethans, D. (Visited: March 2017). Managing schema changes with MongoDB. https://derickrethans.nl/managing-schema-changes.html.

[100] Riak (Visited March 2017). Riak kv. http://basho.com/products/riak-kv/.

[101] Roo, S. (Visited: March 2017). Spring Roo. http://projects.spring.io/spring-roo/.

[102] Ruiz, D. S., Morales, S. F., & Molina, J. G. (2015a). Inferring versioned schemas from nosql databases and its applications. In *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings* (pp. 467–480).

[103] Ruiz, D. S., Morales, S. F., & Molina, J. G. (2015b). Model-driven nosql data engineering. In *Actas de las XX Jornadas de Ingeniería del Software y Bases de Datos (JISBD), (handle: 11705/JISBD/2015/021)* La Laguna, Tenerife, Spain.

[104] Ruiz, D. S., Morales, S. F., & Molina, J. G. (2017a). An mde approach to generate schemas for object-document mappers. In *Proceedings of the 5th International*

Conference on Model-Driven Engineering and Software Development (MODELSWARD) (pp. 220–228).

[105] Ruiz, F. J. B. (2016). *An Approach for Model-Driven Data Reengineering*. PhD thesis, Facultad de Informatica. Universidad de Murcia.

[106] Ruiz, F. J. B., Óscar Sánchez Ramón, & Molina, J. G. (2017b). A tool to support the definition and enactment of model-driven migration processes. *Journal of Systems and Software*, 128, 106 – 129.

[107] Rumbaugh, J. E., Jacobson, I., & Booch, G. (1999). *The unified modeling language reference manual*. Addison-Wesley-Longman.

[108] Rückstieß, T. (2016). `mongodb-schema` npm package. https://www.npmjs.com/package/mongodb-schema. Visited April 2016.

[109] Sadalage, P. & Fowler, M. (2012). *NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley.

[110] Selic, B. (2012). What Will it Take? A View on Adoption of Model-Based Methods in Practice. *Software and Systems Modeling*, 11(4), 513–526.

[111] Simsion, G. (2007). *Data Modeling Theory and Practice*. Technics Publications.

[112] Smith, B. (2015). *Beginning JSON*. Berkely, CA, USA: Apress, 1st edition.

[113] StackOverflow (Visited March 2017). Stackoverflow Data Dump. https://archive.org/details/stackexchange.

[114] Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

[115] Sánchez Ramón, O., Bermúdez Ruiz, F. J., & García Molina, J. (2013). Una valoración de la modernización de software dirigida por modelos. In *Actas de las XIII JISBD* (pp. 288–301).

[116] Tony Clark, Paul Sammut, J. W. (2008). *Applied Metamodelling: A Foundation for Language Driven Development*. Ceteva, second edition edition.

[117] Vaishnavi, V. & Kuechler, W. (2015). Design science research in information systems. http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf. Last access: 21-06-2016.

[118] Wang, K. & Liu, H. (1997). Schema discovery for semistructured data. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), Newport Beach, California, USA, August 14-17, 1997* (pp. 271–274).

[119] Wang, L., Hassanzadeh, O., Zhang, S., Shi, J., Jiao, L., Zou, J., & Wang, C. (2015). Schema Management for Document Stores. In *VLDB Endowment*, volume 8.

[120] Warmer, J. & Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2 edition.

[121] Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE Software*, 31(3), 79–85.

[122] Xtext (Visited March 2017). Xtext documentation. https://eclipse.org/Xtext/documentation/.

[123] Zaharia, M., Chowdhury, M., et al. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.

[124] Óscar Sánchez Ramón (2014). *Model-Driven Modernisation of Legacy Graphical User Interfaces*. PhD thesis, Facultad de Informatica. Universidad de Murcia.