UNIVERSIDAD DE MURCIA

ESCUELA INTERNACIONAL DE DOCTORADO

A Unified Data Metamodel for
Relational and NoSQL databases:
Schema Extraction and Query

Un Metamodelo de Datos Unificado para
Bases de Datos Relacionales y NoSQL:
Extracción y Consulta de Esquemas

D. Carlos Javier Fernández Candel

2022

A Unified Data Metamodel for Relational and NoSQL databases: Schema Extraction and Query

a dissertation presented
by
Carlos Javier Fernández Candel

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
May 2022

This thesis is dedicated to my parents and my brother.

Thanks for keeping the interest rates low on everything I owe you.

Un Metamodelo de Datos Unificado para Bases de Datos Relacionales y NoSQL: Extracción y Consulta de Esquemas

Resumen en Español

Con la aparición de las aplicaciones modernas que hacen un uso intensivo de datos (por ejemplo, Big Data, redes sociales o internet de las cosas), surgieron los nuevos sistemas de base de datos NoSQL (*Not only SQL*) con el objetivo de superar las limitaciones que los sistemas relacionales evidenciaron para soportar tales aplicaciones, como son escalabilidad, disponibilidad, flexibilidad y capacidad para representar objetos complejos.

Estos sistemas de bases de datos se clasifican en varios paradigmas, pero normalmente el término NoSQL se refiere a cuatro de ellos: *columnar*, *clave-valor*, *documentos* y *grafos*. Los sistemas NoSQL de un mismo paradigma pueden tener diferencias significativas en las características y en la estructura de los datos. Esto es debido a que no existe una especificación estándar o teoría que establezca el modelo de datos de un paradigma en particular. Por tanto, en esta tesis asumiremos el modelo de datos de las bases de datos más populares de cada categoría.

En los últimos años, a medida que ha ido aumentando la popularidad de los sistemas NoSQL, ha aparecido el término persistencia políglota (*polyglot persistence*). Este término refiere a sistemas de bases de datos heterogéneos y ha ganado aceptación como la arquitectura de datos del futuro: aplicaciones que utilizan un conjunto de bases de datos que mejor se adaptan a sus necesidades. Hoy en día, las bases de datos relacionales siguen siendo claramente las más utilizadas por un amplio margen, pero los sistemas relacionales más populares están evolucionando para admitir características de NoSQL. Este interés en *polyglot persistence* [111, 98] se debe a dos motivos principalmente: (i) la complejidad y variedad de datos que deben ser administrados por los sistemas de software, y (ii) un solo tipo de sistema de base de datos no satisface todas las necesidades de un número creciente de aplicaciones (por ejemplo, tiendas online, redes sociales o sistemas de gestión del aprendizaje).

Los *modelos de datos* (data models) determinan cómo se pueden organizar y manipular los datos en las bases de datos. Se aplican a un dominio en particular mediante la definición de *esquemas* que expresan la estructura y las restricciones para las entidades del dominio. Dicha información es necesaria para implementar muchas de las herramientas de base de datos. Sin embargo, la mayoría de los sistemas NoSQL son *schemaless* (también conocido como *schema-on-read*), es decir, los datos se pueden almacenar directamente sin requerir la

declaración previa de un esquema. Esta característica está motivada por el hecho de que el ritmo de los cambios en la estructura de datos es considerablemente más rápida en las nuevas aplicaciones de uso intensivo de datos. Por tanto, no hay una comprobación de los datos con el esquema, y los datos de una misma entidad se pueden almacenar con diferentes estructuras, a las que nos referiremos como *variaciones estructurales* (*structural variations*).

La gestión de datos siempre requiere que los desarrolladores o administradores de bases de datos diseñen, creen y desarrollen esquemas de bases de datos. Los desarrolladores siempre deben tener en cuenta el esquema para escribir código, y los administradores deben conocer el esquema para poder realizar tareas comunes como la optimización de consultas. Por lo tanto, las herramientas de base de datos para administrar esquemas son tan esenciales para los sistemas NoSQL como lo han sido para los sistemas relacionales.

Motivación En el informe de Dataversity [18] se expone que la adopción exitosa de los sistemas *NoSQL* requiere herramientas de base de datos similares a las disponibles para los sistemas relacionales. Las herramientas necesitan el esquema lógico de la base de datos para proporcionar funcionalidades comunes de la base de datos como consulta de esquema, visualización, duplicación de datos o generación de código.

Esto implica investigar cómo las herramientas de bases de datos comunes pueden estar disponibles para los sistemas NoSQL. Además, estas herramientas deben construirse teniendo en cuenta el predominio esperado de la persistencia políglota. Por lo tanto, deben admitir bases de datos relacionales generalizadas, así como también bases de datos NoSQL. En el caso de las herramientas de modelado de datos, el cambio hacia soluciones multimodelo (*multi-model*) es evidente: las herramientas de modelado relacional más populares se están ampliando para integrar bases de datos NoSQL (por ejemplo, ErWin y ER/Studio proporcionan funcionalidad para MongoDB) y nuevas han aparecido que soportan una serie de bases de datos relacionales y NoSQL (por ejemplo, Hackolade). La naturaleza multi-model debe tenerse en cuenta para las herramientas de bases de datos, es decir, las herramientas deben admitir múltiples modelos de datos (*data models*).

Las bases de datos *schemaless* no implican que haya una ausencia de un esquema, sino que la información del esquema está implícita en los datos y en el código de las aplicaciones. Por lo tanto, los esquemas implícitos en las bases de datos NoSQL tienen que ser obtenidos mediante ingeniería inversa para construir las herramientas para las bases de datos NoSQL con esta característica. Este proceso de ingeniería inversa debe abordar el hecho de que los sistemas de *schema-on-read* pueden almacenar datos con una estructura diferente incluso perteneciendo al mismo tipo de entidad de base de datos (y tipo de relación en las bases de datos de grafos), es decir, cada entidad o tipo de relación puede tener una o más *structural variation*. Recientemente, se han publicado varios enfoques de extracción de esquemas NoSQL [101, 84, 119, 39], y algunas herramientas de modelado de datos como las mencionadas anteriormente proporcionan algún tipo de funcionalidad de ingeniería inversa.

En la creación de herramientas de base de datos multi-model, la definición de un meta-

modelo genérico, universal o unificado puede proporcionar beneficios [48, 21, 17, 83, 118], debido a que ofrece una visión unificada de diferentes modelos de datos, por lo que sus esquemas se representarán de manera uniforme. Esta uniformidad facilita la creación de herramientas genéricas que son independientes de la base de datos. Con el predominio de las bases de datos relacionales, el interés en las herramientas multimodelo disminuyó y se ha prestado poca atención a la definición de metamodelos unificados. Una propuesta notable es el enfoque DB-Main [48, 77] que definió el metamodelo genérico GER [70] basado en el modelo de datos EER (Extended Entity Relationship) [113]. Más recientemente, se han creado algunos metamodelos universales [17, 83] en el contexto del *Model Management* [21, 20]. Con la aparición de los sistemas NoSQL, se ha propuesto que algunos metamodelos unificados tengan un acceso uniforme a los datos [17, 15], y la idea de un metamodelo unificado para herramientas de modelado de datos se describió en [118].

A veces los datos no están disponibles y es más difícil todavía acceder a las instrucciones de inserción de datos. Debido a esto, las aplicaciones estructuran los datos y controlan la mayoría de las restricciones de los datos, haciendo posible obtener información del código de las aplicaciones como el esquema e incluso referencias entre datos que no se pueden obtener analizando los datos. Por esta razón, el análisis de código puede ser la única forma de obtener el esquema, especialmente en tiempos de desarrollo o evolución de la aplicación. Un trabajo en este área es [86] donde se realiza un análisis de código sobre código Java y se aplica a diferentes versiones del código para comprender la evolución de los datos.

La visualización de los esquemas de bases de datos NoSQL puede no ser práctica, especialmente en aquellas schemaless, debido a que los esquemas pueden tener muchas *structural variations* en sus entidades y, por lo tanto, es conveniente disponer de un lenguaje de consulta de esquemas. En las bases de datos relacionales, este lenguaje puede ser SQL, ya que el estándar SQL-92 especifica cómo se puede representar la información de los esquemas en forma de tablas. Las *structural variations* complican la visualización del esquema NoSQL, en particular si la mayoría de las entidades tienen structural variations o alguna entidad tiene una gran cantidad de ellas. De hecho, se han encontrado miles de variaciones de una entidad en bases de datos en algunos dominios, como DBpedia o biología molecular. El uso de lenguajes de consulta se propone en [119]. Por lo tanto, la utilidad de un lenguaje de consulta de esquemas para esquemas NoSQL es mayor que para los esquemas relacionales.

Esta tesis está motivada principalmente por la falta de un modelo de datos genérico que integre los modelos de datos NoSQL más populares y el modelo de datos relacional. Además, observamos que los enfoques de extracción de esquemas basados en el análisis de código habían recibido poca atención, así como tampoco la automatización de refactoring de bases de datos NoSQL.

Definición del problema y Objetivos Cuando varios paradigmas de bases de datos se vuelven populares, las herramientas de modelado de datos y las herramientas de bases de datos deben admitir todos ellos para que se utilicen ampliamente. Por lo que disponer de un mo-

delo de datos genérico o unificado ofrece la posibilidad de construir herramientas multimodel. Con la creciente adopción de las bases de datos NoSQL y el papel predominante que seguirán jugando las bases de datos relacionales, se hace evidente la necesidad de un modelo lógico unificado, de la misma forma que ocurrió a principios de los noventa cuando surgieron las bases de datos orientadas a objetos y relacionales.

Esta tesis aborda la definición de un modelo de datos unificado con el objetivo de integrar el modelo relacional con los modelos de datos de los cuatro paradigmas más comunes de NoSQL: columnar, documento, clave-valor y grafos. Debido a que no existe una especificación estándar para los modelos de datos NoSQL, se establece una definición para cada paradigma. La definición del modelo de datos unificado debe tener en cuenta: (i) la existencia de structural variations, (ii) la necesidad de diferenciar entre los diferentes tipos de relaciones entre entidades (agregación, referencias, relaciones en grafos y generalización), y (iii) todas las especificidades de las bases de datos NoSQL, como la presencia de tipos de relaciones en las bases de datos de grafos, además de los tipos de entidades.

La construcción de un modelo de datos unificado conlleva definir mappings bidireccionales entre el modelo de datos unificado y cada uno de los modelos de datos de cada sistema de base de datos. La implementación de los mappings es útil para validar el modelo de datos, y también para obtener los extractores de esquema, creados para el sistema más popular de cada paradigma NoSQL: Cassandra y Hbase para columnar, MongoDB para documento, Redis para clave-valor y Neo4j para grafos. Además, los esquemas relacionales de MySQL se han representado en el modelo de datos unificado. En el caso de MongoDB, también se ha creado un extractor de esquemas aplicando un análisis de código.

En cuanto a las aplicaciones del modelo de datos, se ha creado un lenguaje de consulta de esquema genérico, que permite lanzar consultas sobre los esquemas que están representados en el modelo de datos unificado. Los resultados de la consulta se visualizan en una notación visual diseñada e implementada en esta tesis. Además, se explora la posibilidad de utilizar el modelo unificado para definir un lenguaje de consulta de datos genérico. Finalmente, se investiga la automatización del refactoring de bases de datos NoSQL abordando la eliminación de consultas join en una base de datos de documentos. Esto implica analizar el código de las aplicaciones de bases de datos.

En el desarrollo de la tesis se ha utilizado Model-Driven Engineering (MDE) para representar los esquemas en forma de modelos que se ajusten a su metamodelo. De esta forma, los esquemas se representan con un alto nivel de abstracción, lo que facilita su manipulación, en particular con transformaciones de modelos. El modelo de datos unificado se ha creado como un metamodelo y los esquemas son instancias de este metamodelo. Además, se ha definido el lenguaje de consultas basado en un metamodelo.

Por lo tanto, los objetivos de investigación de esta tesis son:

Objetivo 1. Crear un metamodelo unificado (U-Schema: Unified Schema) capaz de representar esquemas lógicos. Este metamodelo debe admitir los tipos más comunes de sistemas NoSQL y relacionales, y debe incluir las siguientes características: (i) la no-

ción de *structural variation*, (ii) los cuatro tipos de relaciones entre tipos de entidad típicos en el modelado de datos lógicos, (iii) la presencia de tipos de relaciones además de los tipos de entidad.

Objetivo 2. Definir los mappings bidireccionales entre U-Schema y los diferentes modelos de datos. Se utilizarán los términos forward and reverse mappings para referirse, respectivamente, a los mappings desde los datamodels a U-Schema y al revés. Se considerará un datamodel por cada sistema NoSQL para establecer los mappings. Se implementarán los forward and reverse mappings para varios sistemas NoSQL, lo que implica tener que extraer primero los esquemas implícitos conforme datamodel específico de ese sistema y luego aplicar los mappings de ese datamodel a U-Schema.

Objetivo 3. Crear un lenguaje genérico para consultar esquemas U-Schema. El lenguaje debe incluir construcciones para expresar fácilmente consultas para buscar información sobre los esquemas extraídos en forma de modelos U-Schema. Al estar definido sobre U-Schema, el lenguaje es independiente de la plataforma. Los esquemas se podrán navegar a través de las diferentes relaciones entre tipos de entidades: agregados y referencias.

Objetivo 4. Diseñar una notación para visualizar esquemas U-Schema. Este objetivo tiene su origen en el anterior ya que los resultados de las consultas deben mostrarse en forma de diagrama. El resultado podría ser un esquema completo o un subesquema.

Objetivo 5. Diseñar e implementar una estrategia de análisis de código estático para descubrir esquemas NoSQL implícitos en el código de las aplicaciones. La estrategia debe ser lo más reutilizable posible. Esto requiere definir representaciones del código y de la información descubierta (operaciones de base de datos y estructuras de datos) que no estén vinculadas a un lenguaje orientado a objetos en partícular ni a una base de datos en concreto.

Objetivo 6. Eliminar de forma automática consultas join en el código. La información extraída en la estrategia de análisis de código del objetivo 5 se utilizará para automatizar un refactoring de base de datos, en particular, eliminar consultas join para mejorar el rendimiento de la aplicación. Eliminar estas consultas conlleva tener que duplicar datos desde el tipo entidad referenciado al tipo entidad que referencia.

Objetivo 7. Explorar la utilidad de U-Schema para definir un lenguaje de consulta de base de datos. Realizar un estudio para identificar los requisitos para crear un lenguaje de consulta universal basado en U-Schema, proponer una posible sintaxis y una posible arquitectura de ejecución.

Por último, la tesis tiene como objetivo desarrollar la infraestructura central para un proyecto a largo plazo del grupo ModelUM. El propósito de este proyecto es construir un

entorno genérico de modelado de datos que integre diferentes utilidades esenciales de bases de datos como: visualización de esquemas, consulta de esquemas, evolución de esquemas, migración de esquemas y generación de datos sintéticos.

Metodología Para lograr los objetivos de la tesis, hemos seguido la metodología Design Science Research Methodology (DSRM) descrita en [97, 115, 120]. En esta metodología se proponen procesos de investigación iterativos organizados en varias etapas o actividades para lograr un objetivo. Las actividades que normalmente constituyen estos procesos son: (i) Identificación del problema y motivación, (ii) Definición de los objetivos de la solución, (iii) Diseño y desarrollo, (iv) Demostración, (v) Evaluación, y (vi) Conclusiones y comunicación. En un proceso DSRM, la actividad de investigación avanza iterativamente, el conocimiento producido en cada iteración se utiliza como retroalimentación para lograr un mejor diseño e implementación del artefacto final.

Discusión de los resultados Se ha definido un metamodelo unificado para la representación de los esquemas lógicos de diferentes bases de datos. Entre sus características más importantes que se diferencian del resto de metamodelos están: (i) proporciona soporte para representar el modelo de datos de los cuatro tipos más comunes de sistemas NoSQL (documento, grafos, clave-valor y columnar) y relacional. (ii) incluye la noción de *structural variation* para representar las diferentes estructuras de datos dentro de un mismo tipo, (iii) incluye tipos de relación para representar las entidades de las referencias en bases de datos de grafos que también incluyen *structural variations*, y (iv) incluyen los cuatro tipos de relaciones entre entidades: agregación, referencias, relaciones en grafos y generalización. Hemos validado el metamodelo inyectando los esquemas de diferentes sistemas de bases de datos.

Se ha establecido la noción de *canonical mapping* en el que existe una correspondencia natural entre cada elemento de un modelo de datos y elementos de U-Schema. Hemos definido formalmente estos mappings desde los modelos de datos relacionales y los cuatro tipos más comunes de sistemas NoSQL a U-Schema (*forward mappings*), y desde U-Schema a los modelos de datos (*reverse mappings*). También hemos establecido la correspondencia entre las diferentes formas de representar los datos en diferentes tipos de bases de datos, por ejemplo, la representación de relaciones con atributos en bases de datos de grafos como agregados en sistemas de documentos, y viceversa. Los mappings se han validado de la misma forma que el metamodelo, implementando la extracción de los esquemas de diferentes tipos de bases de datos.

Se ha implementado una estrategia común para la extracción de esquemas de diferentes tipos de bases de datos y para representar los esquemas en modelos U-Schema implementando los canonical mappings previamente definidos. Las implementaciones se han realizado a través de una serie de map-reduce, algunos de ellos comunes entre ellos, que utilizan una representación común de los esquemas, permitiendo la reutilización de componentes y de los mappings a U-Schema. Hemos validado las implementaciones utilizando una base de

datos común adaptada para cada sistema de base de datos y utilizando una base de datos real de cada tipo. También hemos medido el tiempo de inferencia para cada base de datos. Esta validación ha servido para la validación de la implementación, los mappings y de la adecuación del metamodelo unificado para la representación de los diferentes esquemas.

Se ha desarrollado un proceso MDE de análisis de código para aplicaciones Javascript que utilizan la API de MongoDB para acceder a la base de datos. Hemos creado diferentes metamodelos que pueden ser útiles para representar una gran parte de las sentencias de código de los lenguajes de programación imperativos y orientados a objetos diferentes de JavaScript. El análisis de código se ha utilizado para obtener el esquema de la base de datos producido por la aplicación y se puede aplicar a diferentes versiones de código desde un repositorio de código para obtener la evolución del esquema de la base de datos a lo largo del tiempo. El análisis de código ha permitido encontrar referencias entre objetos de la base de datos que no se podían encontrar con el análisis de los datos almacenados en la base de datos. El análisis de código junto con el esquema ha permitido realizar un refactoring de la base de datos gracias al conocimiento del uso de la base de datos a través de consultas. Hemos validado este proceso con un proceso round-trip en el que desarrollamos un modelo U-Schema junto a una serie de refactorings y después desarrollamos una aplicación que hacía uso de este esquema. Finalmente, aplicamos el proceso de análisis de código y comprobamos ambos modelos comparando el obtenido con el previamente diseñado. También se ha comprobado el refactoring de la base de datos comparándolo con los previamente diseñados.

Se ha creado un lenguaje de consultas de esquemas para poder consultar los modelos U-Schema haciendo uso de las principales características de U-Schema como *structural variations* y los *tipos de relaciones*, permitiendo conocer la estructura de grandes bases de datos con una gran cantidad de *structural variations* fácilmente. Los resultados de las consultas se muestran en una notación gráfica en forma de grafo donde los nodos son los *structural variations* y los arcos son las referencias de U-Schema. Hemos validado este proceso a través de una evaluación en la que han participado expertos del mundo de las bases de datos, quienes han dado su opinión sobre el lenguaje y la notación gráfica creada. También, hemos validado el lenguaje con métricas comúnmente utilizadas en la evaluación de lenguajes y comparándolas con las métricas de otros lenguajes parecidos.

Contribuciones Esta tesis contribuye con un metamodelo lógico unificado (U-Schema) que integra los paradigmas de bases de datos más utilizados: relacional y NoSQL. La creación de este metamodelo ha supuesto también otras aportaciones: (i) La definición de dos modelos lógicos de datos para sistemas NoSQL: uno para sistemas basados en agregación (columnar, documento y clave-valor) y otro para sistemas de grafos. Estos modelos de datos son más complejos que los publicados anteriormente porque incluyen diferentes tipos de relaciones, así como variaciones estructurales, (ii) La especificación formal de mappings bidireccionales entre el metamodelo unificado y los modelos de datos individuales, y (iii) La

definición de una arquitectura con componentes reutilizables para crear un extractor de esquemas para cualquier sistema NoSQL. Los resultados del trabajo relacionado con la definición del metamodelo unificado, la especificación de los mappings y la implementación de los extractores, se han publicado en un extenso artículo de 26 páginas en Information Systems Journal [29].

El metamodelo unificado representa esquemas lógicos de bases de datos, pero en ocasiones es necesario conocer información sobre aspectos físicos como los relacionados con índices o particiones de datos. Por esta razón, los resultados de esta tesis permitieron definir el primer enfoque que aborda la conexión entre esquemas lógicos y físicos. Los resultados obtenidos fueron publicados en el workshop CoMoNoS (Conceptual Modeling for NoSQL data stores) que forma parte de la conferencia Conceptual Modeling 2020 [93]. En este artículo, se presentan los mappings entre el metamodelo U-Schema y un metamodelo físico creado para MongoDB.

En cuanto al trabajo de análisis estático de código, contribuimos con la primera propuesta de extracción de esquemas lógicos del código. Al comienzo de esta tesis, solo Meurice y Cleve [86] habían publicado un trabajo sobre análisis estático de código para aplicaciones NoSQL. Aunque este trabajo se limitó a extraer el union schema de cada colección de MongoDB, y no se abordaron las relaciones ni las structural variations. Nuestra propuesta de análisis de código se realizó en colaboración con el Dr Cleve en una estancia predoctoral en el grupo PRECISE (Universidad de Namur, Bélgica). Además de la extracción del esquema lógico, nuestra estrategia de análisis de código también abordó la automatización de los cambios de esquema (refactoring). Se ha creado una infraestructura con diferentes metamodelos de código para facilitar la detección de las líneas de código a modificar. Hemos aplicado esta infraestructura para implementar la "eliminación de consultas join" para las aplicaciones JavaScript que acceden a bases de datos MongoDB. Los modelos de código y DOS desarrollados son útiles para detectar "code smells" de aplicaciones NoSQL identificados en [19]. Hasta donde sabemos, nuestra propuesta es el primer trabajo sobre la automatización de cambios de esquema NoSQL.

En esta tesis se ha definido la primera propuesta de un lenguaje de consulta genérico de esquemas NoSQL y relacional (SkiQL). SkiQL permite a los desarrolladores expresar consultas sobre esquemas lógicos representados como modelos U-Schema. El artículo que describe SkiQL y la representación gráfica de los resultados de las consultas se ha enviado a Data & Knowledge Engineering, y está disponible en arxiv [53]. Antes de crear SkiQL, experimentamos con el lenguaje Cypher para consultar los modelos U-Schema inyectados en las bases de datos Neo4j, y se utilizó la herramienta Neo4j Browser para visualizar los resultados de la consulta (es decir, grafos). Publicamos esta prueba de concepto en el Congreso Español de Ingeniería del Software y Bases de Datos de 2019 [54].

Se ha realizado un estudio sobre la utilidad de U-Schema para crear un lenguaje genérico para consultar los datos de cualquier tipo de bases de datos NoSQL y relacional. Identificamos los tipos de consultas que serían necesarias y se propuso una sintaxis [29]. También,

se ha realizado una comparación de diferentes metamodelos genéricos para representar esquemas de bases de datos.

El uso de Model-Driven Engineering (MDE) se ha aplicado para implementar todos los enfoques ideados en esta tesis: metamodelo unificado y mappings bidireccionales (es decir, extractores de esquema), el lenguaje SkiQL como DSL y el proceso de ingeniería inversa para el análisis de código. Se realizó una comparación de lenguajes de transformaciones modelo a modelo para elegir la solución más conveniente para la tesis. Este estudio fue presentado en las Jornadas Españolas de Ingeniería del Software y Bases de Datos de 2018 [52]. También, se utilizó y validó la metodología de desarrollo diseñada por el autor de esta tesis [28]. Esta metodología se ha aplicado para desarrollar el enfoque de análisis de código de la tesis.

Agradecimientos

Conocí a mis directores de tesis, Jesús García y Diego Sevilla, cuando estaba en tercero de carrera para hacer mi trabajo de fin grado. Siete años después de aquel momento me encuentro en el final de mi tesis doctoral. Quiero agradecer a mis dos directores todo el apoyo recibido, en todos los aspectos, y todo lo que me han enseñado. En todos estos años me han enseñado sobre conocimientos técnicos e investigación, a redactar y exponer ideas, a impartir clases y evaluar, la participación e incluso dirección de grupos de trabajo y proyectos, también en lo personal y en otros muchos aspectos que valoro considerablemente.

Mis directores de tesis me introdujeron en la investigación desde el primer momento y me ofrecieron todo lo que ha hecho que haya podido redactar esta tesis. Comenzaron proponiéndome un proyecto de investigación junto a una beca colaboración y al finalizar mi carrera, junto con Francisco Javier Bermúdez, me ofrecieron un contrato como investigador en un proyecto de investigación con la empresa OpenCanarias, mientras realizaba mi máster de investigación y trabajo de fin de máster con Jesús y Francisco Javier. Después, gracias a mis directores, sus expedientes y a todo lo que hicimos juntos, conseguimos un contrato de Formación de Profesorado Universitario (FPU) de cuatro años con el que he podido desarrollar la tesis con ellos e impartir clases de Tecnologías de Desarrollo Software que siempre ha sido uno de mis sueños. Quería también dar las gracias a mi profesora de segundo de carrera María Antonia Cárdenas por presentarme a Jesús para iniciar mi carrera investigadora.

Agradecer también el apoyo personal recibido del resto de componentes del grupo de investigación Modelum Group. Grupo que me ha permitido conocer a otros doctorandos como Alberto Hernández y Pablo David Muñoz, que hemos trabajado muy bien juntos, incluso separados por la pandemia y siempre están dispuestos a ayudar. Dar las gracias a Jesús, Francisco Javier y José Ramón Hoyos, por todo el apoyo recibido para impartir las clases desde el primer momento y por resolver mis dudas de profesor.

No me puedo olvidar de mis compañeros de Namur. Agradecer a Anthony Cleve toda la ayuda recibida para poder realizar mi estancia en Bélgica y nuestro proyecto. También agradecer a mis compañeros en Namur: Aboubacar Sylla, Rabeb Abida, Maxime Gobert y Emna Hachicha, por haber hecho de mi estancia en el extranjero una experiencia inolvidable y fácil, gracias a toda su ayuda, tanto en la investigación como en todo lo personal.

Me gustaría agradecer a mi hermano mayor el haberme descubierto esta pasión por la informática que compartimos, y a mis padres por haber hecho muy fácil poder recorrer todo el trayecto de mi vida hasta esta tesis. Por último, quería agradecer a todos mis alumnos que he tenido durante los tres años que he impartido docencia por haber sido los mejores alumnos que pueden tener un profesor que comienza su carrera en la docencia.

A Unified Data Metamodel for Relational and NoSQL databases: Schema Extraction and Query

Abstract

The Database field is undergoing significant changes. Although relational systems are still predominant, the interest in NoSQL systems is continuously increasing. In this scenario, polyglot persistence is envisioned as the database architecture to be prevalent in the future. Therefore, database tools and systems are evolving to support several data models.

Most NoSQL systems are schema-on-read: data can be stored without first having to declare a schema that imposes a structure. This schemaless feature offers flexibility to evolve data-intensive applications when data frequently change. Such an absence of schema declaration makes *structural variability* possible, i.e., stored data of the same entity type can have different structure. Moreover, data relationships supported by each data model are different; For example, document stores have *aggregate objects* but not *relationship types*, whereas graph stores offer the opposite. However, freeing from declaring schemas does not mean its absence, but rather they are implicit in data and code, and they must be designed and evolved by developers and administrators. Therefore, tools similar to those available for relational systems are also needed to help understand NoSQL schemas.

Multi-model database tools normally use a generic or unified logical metamodel to represent schemas of the data models that they support. Such metamodels facilitate developing database utilities, as they can be built on a common representation. Also, the number of mappings required to migrate databases from a data model to another is reduced. *Structural variability* complicate NoSQL schemas, in particular if most of entities have variations or some entity has a large number of variations. Therefore, NoSQL schema management tools will have three main components: schema extraction, schema query and schema visualization.

In this thesis, we present the U-Schema unified metamodel able to represent logical schemas for the four most popular NoSQL paradigms (columnar, document, key-value, and graph) and relational schemas. We will formally define the mappings between U-Schema and the data model defined for each database paradigm. How these mappings have been implemented and validated for the schema extraction will be discussed. Also we present our static code analysis approach of applications for the schema extraction and database refactoring. Finally, we present the SkiQL generic schema query language aimed to query U-Schema schemas and a graphical notation to visualize U-Schema schemas.

Contents

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Problem statement and Goals . . . 6
- 1.3 Research methodology . . . 9
- 1.4 Outline . . . 13

2 Background 15
- 2.1 Model-Driven Engineering . . . 15
- 2.2 The User Profiles Running Example . . . 19
- 2.3 Data Modeling . . . 20

3 State of the Art 31
- 3.1 Generic Schema Metamodels . . . 31
- 3.2 Schema Extraction . . . 46
- 3.3 Code Metamodels and Code Analysis . . . 56
- 3.4 Schema Query and Visualization . . . 66

4 The U-Schema Unified Metamodel 69
- 4.1 Concepts in the U-Schema unified data model . . . 69
- 4.2 The U-Schema Metamodel . . . 71
- 4.3 U-Schema Flavors: Full Variability vs. Union Schema . . . 74
- 4.4 Mappings between U-Schema and the Logical Data Models . . . 75
- 4.5 Applications of the U-Schema Metamodel . . . 77

5 Strategies for the Extraction of U-Schema Models 87
- 5.1 A Common Strategy for the Extraction of U-Schema Models . . . 87
- 5.2 Graph Model Schema to U-Schema Models . . . 90
- 5.3 Document Model Schema to U-Schema Models . . . 94
- 5.4 Key-Value Model Schema to U-Schema Models . . . 100
- 5.5 Columnar Model Schema to U-Schema Models . . . 103
- 5.6 Relational Model Schema to U-Schema Models . . . 104
- 5.7 Validation of the Schema Building Process . . . 108

6 Extracting U-Schema logical schemas from code of database applications 111
- 6.1 Overview of the approach . . . 111

6.2 Obtaining an abstract representation of the code . . . 118
6.3 Discovering the database schema . . . 126
6.4 Generating join removal plans . . . 135
6.5 Validation . . . 140

7 A Generic Schema Query Language: SkiQL **145**
7.1 Kinds of NoSQL Schemas . . . 145
7.2 Visualization of Complete Schemas . . . 146
7.3 SkiQL Query Language: Syntax and Semantics . . . 150
7.4 Implementation of SkiQL . . . 159
7.5 Evaluation . . . 161

8 Conclusions and Future Work **167**
8.1 Discussion . . . 168
8.2 Contributions . . . 172
8.3 Future Work . . . 175
8.4 Publications, Projects, and Grants . . . 177

References **181**

List of Figures

1.1 Design Science Research Methodology Process. 9

2.1 A simple example of metamodel: Entity metamodel. 17
2.2 "User profile" running example schema. 20
2.3 Graph Data Model. 22
2.4 User Profiles Graph Database Example. 23
2.5 Document Data Model. 24
2.6 "User Profiles" Document Database Example. 25
2.7 Key-Value Database Example for running example. 26
2.8 Columnar database example for the running example. 28
2.9 "User Profile" relational example. 29

3.1 The new aspects of the Generic Entity/Relation (GER) metamodel. 33
3.2 The Meta-Construct of the Supermodel of Atzeni et al. (Extracted from [17]). 35
3.3 The GeRoMe metamodel (Extracted from [83]). 37
3.4 The SOS Metalayer of Atzeni et al. (Taken from [15]). 37
3.5 The NoSQL Abstract Model (NoAM) metamodel. 39
3.6 The ERwin Logical/Conceptual Metamodel (Taken from [118]). 39
3.7 The TyphonML Metamodel (Taken from [114]). 40
3.8 The PartiQL data model. 42
3.9 Schema extraction process of Klettke et al. (taken from [84]). 48
3.10 Structure identification graph of Klettke et al. (taken from [84]). 50
3.11 Documents Examples of Wang et al. (Extracted from [119]). 52
3.12 Canical Form example of Wang et al. (Extracted from [119]). 53
3.13 eSiBu-Tree example of Wang et al. (Extracted from [119]). 54
3.14 Excerpt of the Code Package of the KDM Metamodel (Extracted from the specification in [68]). 57
3.15 Excerpt of the Action Package of the KDM Metamodel (Extracted from the specification in [68]). 58
3.16 Excerpt of the Java Modisco Metamodel (Extracted from [90]). 59
3.17 Main metaclasses of FAMIX Metamodel (taken from its specification [46]). 60

4.1 U-Schema Metamodel. 73
4.2 Overview of a generic query architecture based on U-Schema. 80

5.1 Generic Schema Extraction Strategy. 88

5.2 Person Aggregates Address in Neo4j. 93
5.3 User Profiles Complete Schema for Graph Stores. 95
5.4 User Profiles Union Schema for Graph Stores. 96
5.5 Example of Application of the Reverse Mapping from a `RelationshipType` of U-Schema to a Document Schema. 99
5.6 User Profiles Complete Schema for Document Stores. 101
5.7 User Profiles Union Schema for Document Stores. 102
5.8 Inference to Query time ratio. 109

6.1 Users and Movies objects in the "streaming service" store. 112
6.2 Entities and Queries extracted for the FWM example. 115
6.3 Overview of the U-Schema extraction and join query removal approach. . 117
6.4 Excerpt of the main elements of the Code Metamodel. 118
6.5 Excerpt of the variable related metaclasses of the Code Metamodel. 119
6.6 Excerpt of the Statements metaclasses of the Code Metamodel. 120
6.7 Excerpt of the *Code* model extracted for the running example (starting from line 9). 122
6.8 Control Flow Metamodel. 123
6.9 *Control Flow* model for the the running example. 124
6.10 Graph excerpt of the Control Flow model for the running example. 127
6.11 Database Operation&Structure Metamodel. 128
6.12 Nodes and edges in Control Flow models. 129
6.13 Database Operations & Structure Model for the running example. 133
6.14 U-Schema model resulting of the code analysis. 135
6.15 Validation designed schema. 141

7.1 Running example for aggregate and graph stores. 147
7.2 U-Schema complete schema for "UP-aggregate." 149
7.3 U-Schema complete schema for "UP-graph." 150
7.4 The complete schema of union types for "UP-aggregate" schema. 150
7.5 The `User` entity type variation subschema for "UP-aggregate." 153
7.6 The `Watchedmovies` relationship type variation subschema for "UP-graph." 153
7.7 Subschema returned for queries Q2 and Q5 on "UP-aggregate" schema. . . 156
7.8 QR query examples. 156
7.9 Subschema returned for query Q4 on "UP-aggregate" schema. 156
7.10 Subschema returned for query *Q6* on "UP-aggregate" schema. 157
7.11 Subschema returned for query *Q7* on "UP-graph" schema. 158
7.12 Subschema returned for query *Q8* on "UP-graph" schema. 159
7.13 The SkiQL metamodel for the definition of the abstract syntax of the language. 160
7.14 An overview of the visualization process and its implementation. 161

List of Tables

1.1 Types of NoSQL systems and some example implementations. 3

3.1 Set of 13 Meta-constructs defined in [17]. 36
3.2 The most important roles of the GeRoMe metamodel [83]. 38
3.3 Comparison of approaches defining a Generic Metamodel. 47

4.1 Mapping between logical modeling concepts and NoSQL/Relational Database Systems. 72

5.1 Database Sizes. 109
5.2 Times for inference and queries for all the database implementations. . . 110

6.1 DOS metamodel to U-Schema metamodel Mappings. 134

7.1 Mapping between U-Schema and graphical notation. 149
7.2 SkiQL, GraphQL, Cypher and SPARQL metrics. 162
7.3 Questionnaire results. 165

1 Introduction

With the advent of modern data-intensive applications (e.g., Big Data, social networks, or IoT), NoSQL (*Not only SQL*) database systems emerged to overcome the limitations that relational systems evidenced to support such applications, namely, scalability, availability, flexibility, and ability to represent complex objects. These systems are normally classified in four categories or paradigms according to the data model: columnar, document, key-value, and graph [98].

Although the popularity of NoSQL systems increased over last years,* relational systems are still predominant today, and new high-performance architectures are appearing for relational systems ("New SQL" movement). However. the idea of "one size does not fit all" has been gaining acceptance and *polyglot persistence* (a new term coined for heterogeneous database systems) is considered the data architecture of the future [111]: applications using the set of databases that better fit their needs. This is supported by the growth of multi-model systems: the first eight databases in the DB-engines ranking† are multi-model, and the main relational database vendors are including support of NoSQL systems in their products. Two facts have mainly motivated the interest in *polyglot persistence* [111, 98]: (i) the

*Three of the top 10 being NoSQL systems in the DB engines ranking (https://db-engines.com/en/ranking) as of May 2022: MongoDB (5th), Redis (6th), and Elasticsearch (8th).

†https://db-engines.com/en/ranking.

complexity and variety of data to be managed by software systems, and (ii) a single type of database system does not fit all the storage needs of an growing number of increasingly complex systems (e.g., learning management systems, online retail systems, or social networks.)

Data models determine how data can be organized and manipulated in databases. They are applied to a particular domain by defining *schemas* that express the structure and constraints for the domain entities. Such information, provided by schemas, is necessary to implement many database tools. However, most NoSQL systems are *schemaless* (a.k.a *schema-on-read*), that is, data can be directly stored without requiring the previous declaration of a schema. This feature is motivated by the fact that the pace of data structure changes is considerably faster in the new data-intensive applications. Therefore, there is no data checking against the schema, and data of the same entity can be stored with different structures, which we will refer as *structural variations*.

Being *schemaless* does not mean the absence of a schema, managing databases always requires to design, create and evolve database schemas by developers or database administrators (DBA). Developers must always have in mind the schema to write code, and administrators have to know the schema in order to perform common tasks as optimizing queries. Therefore, database tools to manage schemas are as essential for NoSQL systems as they have been for relational systems [18].

NoSQL systems of the same paradigm can have significant differences in features and in the structure of the data. This is because there is no specification, standard, or theory that establishes the data model of a particular paradigm. In this thesis, we will assume the data model of the most popular stores of each category. Table 1.1 shows the main features of the four mentioned NoSQL paradigms.

The four categories of NoSQL stores can be classified in two groups depending on the kind of relationship that is prevalent to organize data. In "aggregate-based data models," databases store semi-structured data which form aggregation hierarchies, and references are established by using object identifiers. Columnar, Document, and Key-value systems are based on aggregation. Instead, graph-based systems are organized as a graph of objects connected through arcs denoting a binary relationship between the entity types to which connected objects belong, and aggregation is normally not supported. In aggregate-based

NoSQL System Types			
Type	**Data structure**	**Appropriateness**	**Database Systems**
Key-value	Associative array of key-value pairs	Frequent small read and writes with simple data	Redis, Memcache
Columnar	Tables of rows with varying columns. Column-based physical storage	High performance, availability, scalability, and large volumes of data for OLAP queries	HBase, Cassandra
Document	JSON-like document collections	Nested Objects, structural variation, and large volumes of heterogeneous data	MongoDB, Couchbase
Graphs	Data connected in a graph	Highly connected objects, references prevail over nested objects	Neo4j, OrientDB

Table 1.1: Types of NoSQL systems and some example implementations.

systems, objects stored are instances of entity types, while both entity and relationship types can be instantiated in graph-based systems.

The purpose of this thesis is to define a generic or unified metamodel able to integrate NoSQL and relational data models, and establish the bidirectional mapping between each individual data model and the unified metamodel. Regarding the extraction of schemas, reverse engineering strategies have been devised to analyze stored data and database code. Also, applications of the generic metamodel has been investigated.

The rest of this introduction chapter is organized as follows. First, we motivate the work in Section 1.1. Then, we state the problem and the goals of the thesis in Section 1.2. Later, the research methodology is described in Section 1.3. Last, the structure of the rest of the thesis in Section 1.4.

1.1 Motivation

The DataVersity report [18] exposes that the successful adoption of *NoSQL* systems requires database tools similar to those available for relational systems. Tools that requires the database schema in order to provide common database functionalities like schema query, visualization, database refactoring, or code generation. This entails a research effort to study how these utilities can be developed for each of the NoSQL data models. In addi-

tion, the tools should be built taking into account the expected predominance of polyglot persistence. Thus, they should support widespread relational databases as well as NoSQL databases, i.e. they should support multiple data models. In fact, the shift of data modeling tools towards multi-model solutions is evident: the most popular relational modeling tools are being extended to integrate NoSQL stores (e.g., ErWin[‡] and ER/Studio[§] provide functionality for MongoDB) and new tools supporting a number of relational and NoSQL databases have appeared (e.g., Hackolade[¶]). This multi-model nature should be considered in the most of database tools.

Most of NoSQL systems do not force to declare a schema prior to manage a database, but schemas are implicit in code and stored data. Therefore, they must be reverse engineered always that the development of a database tool requires to know the schema information at run-time in order to implement some functionality, e.g., schema visualization, database refactoring, or code generation. This reverse engineering process must tackle the fact that *schema-on-read* systems can store data with different structure even belonging to the same database entity type (and relationship type in graphs), i.e., each entity or relationship type can have one or more *structural variations*.

Several NoSQL schema extraction strategies have been published. They analyze stored data to discover the database schema, and most of them only consider the document data model as reflected in the MongoDB system[‖] [84, 101, 119]. Schemas can also be derived from data insert scripts as shown in [39], where an approach is proposed for Neo4j stores. Analyzing database code is a third alternative to obtain NoSQL schemas. Query, constraints and code using query results are an information source to discover the implicit schema. Code analysis is necessary to automate code updating when the data structure evolves. A code analysis approach to discover how schemas evolve along time was proposed by Loup Meurice and Anthony Cleve [86] for MongoDB stores and different versions of Java applications. Also, some data modeling tools, as those mentioned above, provide some kind of reverse engineering functionality based on the stored data or data insert statements.

When building multi-model database tools, the definition of a generic, universal, or uni-

[‡]Erwin website: http://erwin.com/products/erwin-data-modeler.
[§]ER/Studio website: https://www.idera.com/er-studio-enterprise-architecture-solutions.
[¶]Hackolade website: https://hackolade.com/.
[‖]MongoDB website: https://www.mongodb.com/.

fied metamodel can provide some benefits [48, 21, 17, 83, 118]. It offers a unified view of different data models, so that their schemas will be represented in a uniform way. This uniformity facilitates building generic tools that are database technology agnostic. With the predominance of relational databases, interest in multi-model tools decreased in the late 1990s, and little attention has been paid to the definition of unified metamodels. A remarkable proposal is the DB-Main approach [48, 77] that defined the GER generic metamodel [70] based on the EER (Extended Entity Relationship) data model [113]. More recently, some universal metamodels [17, 83] have been created in the context of Model Management [21, 20]. With the emergence of NoSQL systems, some unified metamodels have been proposed to have a uniform access to data [17, 15], and the idea of an unified metamodel for data modeling tools was outlined in [118]. More recently, the creation of a generic data model has been announced for Hackolade, but it has not been presented yet.

Unified data models can also be used to define data query generic language. Three decades ago, the ODMG standard [31] specifies a unified object-oriented data model along with a SQL-like language to make it easier for programmers to use object-oriented databases. Currently, the growing use of NoSQL and other kinds of data stores, at the same time that relational systems continue being predominant, is motivating the definition of some generic data models on which generic query languages are defined. For example, Amazon has recently developed the PartiQL data model and query language [7] to offer uniformity in the treatment of the variety of data models (relational, document, columnar and key-value) and data processing engines (NoSQL, relational, and data lakes) used in the company. At this moment, PartiQL does not integrate graph data models. It should be noted that PartiQL was available after the inception of this thesis.

Diagramming is not practical when schemas include many entities (e.g., tables in a relational schema), and a schema query language is then convenient. In relational databases, this language can be the proper SQL since that the standard SQL-92 specifies how information on schemas could be represented in form of tables. In the case of schemaless NoSQL stores, *structural variations* can complicate the visualization of a readable schema, in particular if most of entities have variations or any entity has a large number of variations. In fact, thousands of variations for a database entity have been found in datasets of some domains, such as DBpedia or molecular biology, so that using query languages is suggested in [119].

In [74], a visual notation was proposed for NoSQL schemas, and that work evidenced that diagramming is not useful when information on variations is desired, although the number of variations is small, and therefore a query language is essential to inspect NoSQL schemas.

Our thesis work was mainly motivated by the lack of a generic data model that integrates the most popular NoSQL data models and the relational model. Also, we observed that schema extraction approaches based on code analysis had received little attention, as well as automating database refactorings for NoSQL systems.

1.2 Problem statement and Goals

When several database paradigms are popular, data modeling tools and database tools should support all of them to be widely used. Then, a generic or unified data model offer a convenient solution to build multi-model tools. With the increasingly adoption of NoSQL systems and other kinds of storage, and the predominant role that relational systems will continue playing, the need of a unified logical model is evident, in the same way as occurred in the early nineties when object-oriented and object-relational databases emerged. This thesis has tackled the definition of such a generic data model with the goal of integrating the relational model with the data models of the four main NoSQL paradigms: columnar, document, key-value, and graph. Since there is no standard specification for NoSQL data models, a definition has to be established for each paradigm. The definition of the unified data model should have into account (i) the possible existence of structural variations of the entity types, (ii) the need of differentiate between different types of relationship between entities, and (iii) all the specificity of NoSQL stores, as the presence of relationship types in graph stores, in addition to entity types.

Building a unified data model entails to define the bidirectional mappings between the unified data model and each of the individual data models. The implementation of the mapping from data of a database system to the unified data model is useful to validate the data model, and also to have the schema extractors. In the thesis, extractors are created for the most popular system of each NoSQL paradigm: Cassandra and Hbase for columnar, MongoDB for document, Redis for key-value, and Neo4j for graph. Moreover, MySQL relational schemas have been represented with the unified data model. In the case of MongoDB, an schema extractor has been also created by applying a code analysis.

Regarding the applications of the data model, a schema query generic language is created, which allow queries to be issued on schemas that are represented in the unified data model. Query results are visualized in a visual notation designed and implemented in this thesis. Also, the usefulness of the unified model to define a generic query language is explored.

Finally, the automation of NoSQL database refactorings has been investigated by tackling the removal of join queries in a document database. This involves to analyze code of database applications.

For the development of the work, Model Driven Engineering (MDE) technology is applied to represent schemas in form of models that conforms to its metamodel. In this way, schemas are represented at a high level of abstraction, which makes it easier their manipulation, in particular model transformations. The unified data model will be created as a metamodel, and schemas are instances of this metamodel. Also, a metamodel-based language definition workbench has been used to create the schema query language.

Therefore, the **research goals** of this work are as follows:

Goal 1: Create a unified metamodel able to represent logical schemas. This metamodel must support both the most common kinds of NoSQL systems and relational systems, and and include the following features: (i) the notion of *structural variation* for entity and relationship types, as most NoSQL systems are schemaless; and (ii) the four kinds of relationships between entity types that are typical in logical data modeling: aggregation, references, graph relationships, and generalization; (iii) the presence of relationship types in addition to entity types. This metamodel has been called U-Schema (**U**nified Schema).

Goal 2: Define the bidirectional mappings between U-Schema and the different individual data models. The terms *forward* and *reverse* mappings will be used to refer, respectively, the mapping from individual data models to U-Schema and the opposite one. A data model will have to be determined for each considered NoSQL system in order to establish the mappings. Forward mappings will be implemented for several NoSQL systems, which implies to first extract implicit schemas in form of the data model specified for that system, and then apply the mapping of that model to U-Schema.

Goal 3: Build a generic language aimed to query U-Schema schemas. This language must include constructs to easily express queries to find information on the schemas extracted in form of U-Schema models. Being defined on U-Schema, the language would be platform-independent. Schemas could be traversed through relationships between entity types: aggregates and references.

Goal 4: Design a notation to visualize U-Schema schemas. This goal is originated by the previous one as query results should be shown in form of diagram always that their size allows it. The result could be a complete schema or a subschema.

Goal 5: Design and implement a code analysis strategy to discover implicit NoSQL schemas in the application code. The strategy should be as reusable as possible. This requires to define representations of the code and the information discovered (database operations and data structure) which are not tied to a particular object-oriented language and database.

Goal 6: Automating the removal of join queries. The information extracted by means of the code analysis strategy that constitutes the goal 5 will be used to automate a complex database refactoring, in particular the removal of join query to improve the performance of queries. This removal entails duplicate data of the referenced entity type in the referencing entity type of the join.

Goal 7: Explore the usefulness of U-Schema to define a database query language. Perform a study to identify the requirements to create a universal query language based on U-Schema, and show query examples to illustrate the different kinds of queries.

It is worth noting that the thesis is aimed to develop the core infrastructure for a long-term project of ModelUM group. The purpose of this project is to build a data modeling generic environment that integrates different essential database capabilities as: schema diagramming, schema query, schema evolution, schema migrators, and synthetic data generation. The motivation to create this environment can be found in the conclusions of several reports, as "Insights into Modeling NoSQL" [18]: Schemas and data modeling are necessary for NoSQL stores, and their successful adoption demands to build a database tooling simi-

lar to that available for relational databases. But, the polyglot persistence has increased the need to apply the same techniques to different kind of databases in a uniform way.

1.3 Research methodology

Design science (DS) is a research paradigm focused on the building of artifacts that are designed to help human beings in a particular task [106]. That is, the artifacts are designed and implemented to achieve a concrete goal. Computer Science is a discipline where DS is applicable in most of its areas. In fact, several design science research methods (DSRM) have been published for Information Systems and Software Engineering [97, 115, 120]. These methods propose iterative research processes organized in several stages or activities to achieve the goal. Figure 1.1 shows the activities that normally constitute these processes: (i) Identification of the problem and motivation, (ii) Definition of the objectives of the solution, (iii) Design and development, (iv) Demonstration, (v) Evaluation, and (vi) Conclusions and communication.

In a DSRM process, the research activity iteratively progresses: the knowledge produced in each iteration is used as feedback to achieve a better design and implementation of the final product.

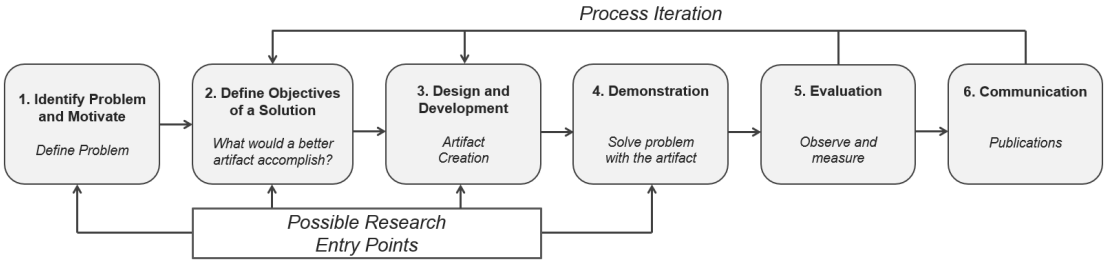

Figure 1.1: Design Science Research Methodology Process.

Next, it will be explained how DS has been applied in this thesis. Regarding to the problem addressed, it was identified with the experience gained in a previous thesis supervised by the same advisors of this thesis [51]. In that work, a metamodel to represent aggregate-based databases was proposed along with a schema extraction process for MongoDB stores. Since a polyglot persistence scenario is imposing as future trend for databases [111], this thesis was conceived to convert the previous metamodel in a generic metamodel, and build

extractors for the main store of each NoSQL paradigm. Diagramming schemas is crucial to provide developer a visual representation of schemas that makes it easier its understanding. Thus, a definition of NoSQL logical schema along with a notation to visualize such schemas were presented in the Severino Feliciano's thesis [51] and a paper of Alberto Hernandez et al. [74]. Both works concluded the convenience of defining a schema query language for large schemas, in the same way that SQL can be used to query relational schemas. Also, code analysis approaches to extract NoSQL logical schemas had not been published, and automating database refactoring there was not received attention.

In the previous sections of this chapter, we motivated the problem and defined the goals of the thesis. The goals basically are: (i) Define the U-Schema unified metamodel able to represent NoSQL y relational schemas; (ii) Define and implement the bidirectional mappings between U-Schema and the integrated data models; (iii) Create the SkiQL schema query language; (iv) Design and implement a graphical notation for U-Schema models; (v) Design and implement an code analysis approach to infer schemas from code of NoSQL database applications; (vi) Automate a non trivial database refactoring: remove join queries. (vii) Explore the definition of a data query language based on U-Schema.

We started by defining the unified metamodel. We tried to extend the metamodel proposed for document data model in [51], but the difficulties encountered motivated to create the metamodel from scratch. Once created, we expressed the forward and backward mappings, which serve to test the metamodel. As we wrote mapping rules we found details to be modified in the metamodel. Then schema extractors were built for a system of each paradigm. This implied to define a data model for that system, to implement an extraction of schemas for the data model, and then to implement the forward mapping between that data model and U-Schema. This allowed to validate the set of mappings.

Extractors were developed as follows. Firstly, we added the implementation of the forward mapping to the existing extractor for document stores, in particular MongoDB. In addition, we rewrite the map-reduce operation of the process to be multi-threading. Then, we decided to tackle the inference of schemas for the Neo4j graph store, because document, key-value and columnar are aggregate-based systems, but graph systems are a completely different database data model where references are prevalent. Building this extractor, some errors in U-Schema were detected and modified, which concerns to aspects related to graph

stores. Also, we realized that the inference process to discover the schema for document and graph data models differed slightly. Therefore, we established a general strategy, and adapted both processes to this strategy. Next, we built the extractors for columnar and key-value stores. At this point, we observed that if a particular pattern is followed to record aggregation in these two systems, a single data model can be defined to represent schemas of columnar, document, and key-value stores. Redis and Hbase was the stores chosen for key-value and columnar, respectively. For columnar, we also developed the inference process for Cassandra because schemas must be declared for Cassandra stores, and we could easily validate the obtained schemas by comparing then with the schemas available. We also tackled the extraction for the MySQL relational system. The definition of the U-Schema metamodel and the development of the extractors take about 14 months, and the elaboration of the paper that present the results takes about 4 months. It is worth noting that the task of analyzing the applicability of U-Schema to define a data query generic language was requested by the editor of the journal where the paper mentioned above was published, which took about 3 months.

The schema extraction from code was chosen the task to be started during the 3-months predoctoral stay in the Precise group of the Namur University. The code analysis strategy to extract logical schemas was implemented as a 3-steps model transformation chain. In the first step, the code is represented by using two metamodels: Code and Control Flow. First, source code is injected in a model that represents statements of object-oriented languages. Next, the model that represents the control flow is obtained from the Code model, which hold references to the former. In the second step, the control flow model is traversed to generate a model that represents the CRUD database operations and the data physical structure, and the relationships between operations and data structures. For this, the Database Operations and Structure (DOS) metamodel was create. In the third step, the U-Schema logical schema is obtained from the physical schema that is part of the DOS model. Afterward, we investigated how DOS models could also be useful to automate NoSQL database refactorings. In particular, we tackled to discovery join queries in order to provide database administrators information helpful to decide whether a determined join query should be removed by duplicating data accessed on the referenced entity into the referencing entity. Once database administrators choose a join query to be removed, the schema change oper-

ation is applied, that is, schema, data and code are automatically updated. Each step of the process consisted in a complex model-to-model transformation that was implemented and tested following the methodology presented in [28], in which each transformation is tested before to validate the complete process. At this moment, we have prepared a paper where we explain the process of static code analysis to obtain the schema and refactoring of the database.

Regarding the SkiQL schema query language, we first created a solution that used the Cypher language [42] to express queries [54]. For this, unified schemas were represented as Neo4j graph databases [95]. A a model-to-text transformation was created to obtain the Neo4j schema, which generates INSERT operations from U-Schema models. A visual representation was designed to graphically show query results in the Neo4j browser [5]. Analyzing this solution, we observed that Cypher queries were difficult to write and not concise. Then, we decided to build a language tailored to express queries on U-Schema schemas, which were translated to Cypher to be executed on Neo4j graph schemas. However, we detected that Neo4j browser diagrams were difficult to understand. This latter was mainly due to that properties had to be represented as nodes connected to its entity type. At this point, we decided to design and implement a graphical notation from scratch, and discard the use of the graph database. The development of all this tools took around 12 months, including first solution based on Neo4j, the new visualization and the design and implementation of the language. This work resulted in a paper sent to Data & Knowledge Engineering, the pre-print version can be found in arxiv [53],

In this thesis, MDE has been used as implementation technology: (i) U-Schema is an Ecore/EMF metamodel [110] and extractors manage schemas in form of models; (ii) a model transformation chain has been built to implement the extraction of schemas from database code; and (iii) a metamodel-based DSL definition workbench has been used to create the concrete syntax of SkiQL, and its engine manages schemas represented as models. Using MDE we have achieved the benefits exposed in [102], where it was analyzed the application of MDE techniques to relational data engineering.

Finally, we will comment how the tools created were evaluated. The extractors were first validated by creating several synthetic databases, and after we used databases injected from real datasets. In the case of SkiQL, some language metrics defined in [41] were calculated

and the survey mentioned above was carried out. To validate the schema extraction from code, we built a small Web application that managed a MongoDB database whose schema had previously been established as a U-Schema model, we then launched the reverse engineering process to extract the schema, and we compared the obtained schemas with the previously defined schema. The code analysis process has also been tested in the different stages of the chain as commented above: Code models were validated by generating code that was compared with the original code (small code fragments); Control Flow models were validated by creating Neo4j graphs, and visually inspecting these graphs to check whether or not they match the control flow in code scripts. Finally, DOS models were manually checked against the corresponding code script to see whether or not data structure and queries were correctly represented.

1.4 Outline

The structure of the this thesis is as follows:

- **Chapter 2** introduces the background needed to understand the following sections. Here, some aspect of the Model-Driven engineering and database data models are presented.
- **Chapter 3** shows an analysis of the current state of the art of works in schema generic metamodels, NoSQL schemas inference, metamodels to represents application soruce code, schema queries and schema visualization.
- **Chapter 4** describes the unified metamodel U-Schema defined in deep. The most relevant and differentiating aspects are exposed.
- **Chapter 5** describes in detail the bidirectional mappings between U-Schema and the different individual data models, the common reverse engineering strategy and the implementation and validation for each kind of databases.
- **Chapter 6** describes the code analysis approach to extract the schema and the refactoring of join queries. The metamodels defined to represent the code and the static code analysis algorithms are also presented.

- **Chapter 7** describes the schema query language developed and the implemented visualization of the U-Schema schemas.
- **Chapter 8** we stated the achievement of the goals defined, we expose the contributions and the publications of the results of the thesis and we present future work.

2

Background

This chapter introduces the background needed for the better understanding of this thesis. First, we will describe the data models for the different database paradigms addressed in our work. Next, we will introduce the basics of Model-Driven Software Engineering (MDE), and after, we will define the data models integrated in our unified metamodel. Before of this data model description, we will introduce a conceptual schema example which be used to illustrate each of the data models.

2.1 Model-Driven Engineering

Model-Driven Software Engineering (*Model-Driven Engineering*, *MDE*) is a discipline within Software Engineering that deals with the systematic use of software models to improve productivity and other aspects related to software quality, such as evolution and interoperability between systems. Since MDE emerged at early years of this century, this vision of software construction has shown its potential to automate most tasks involved in the software development life cycle, both in forward engineering and reverse engineering.

MDE is based on four main foundations: (i) *metamodels* are used to represent the structure of software aspects as code, data, behavior, or tests , or also application domains. (ii) *models* are instances of metamodels, which represents an use for a particular scenario, e.g. the

control flow of a Java program, a state machine of an automatic teller machine, or a database schema of a MongoDB store; (iii) Domain-Specific languages (DSL) are created to provide notations with which to express the models, and (iv) model transformations are used to automate software development tasks We will briefly introduce these principles below.

2.1.1 Metamodeling

A metamodel is a formalism that describes concepts and relationships that describes a particular software aspect or domain of interest. Just as grammars describe the structure of valid programs, metamodel describe the structure of valid models. For example, a UML state machine metamodel determines the structure of any UML state diagram. A metamodel is normally defined as an object-oriented conceptual model expressed in a metamodeling language. Metamodeling languages provide four main elements for expressing metamodels:

- Classes (also called metaclasses) to represent domain concepts.
- Attributes to represent properties of the domain concepts, whose values are primitives types or arrays.
- Aggregations and references between pairs of classes to represent relationships between concepts.
- Generalizations between child classes and parent classes to represent specializations between domain concepts.

An aggregation from *A* class to *B* class means that *B* instances are part of *A* instances, while a reference from *A* to *B* means that *A* instances hold some kind of reference to *B* instances.

Eclipse Modeling Framework(EMF) [110] is the more widely used platform to apply MDE. Its core element is the Ecore metamodeling language, and EMF integrates a set of tools aimed to tasks as graphically defining metamodels, creating the concrete syntax of DSLs, or writing model transformations. Ecore includes the elements indicated above to define metamodels.

Typically, a subset of the UML Class diagram notation is used to represent metamodels because they are similar a domain class models. For example, figure 2.1 shows a Ecore metamodel for expressing business entity models. This metamodel includes the root class *EntityModel* that aggregates a set of one or more *Entities* (relation *entities*), and each entity aggregates, in turn, zero or more *Properties* (relation *props*) and zero or more *Relationships* (relation *rels*). *Entity*, *Property* and *Relationship* classes inherit from *NamedElement* that includes the *name* attribute. A property also has a data type (attribute *type*) and a relationship has a *cardinality* enumeration attribute and a *composite* boolean attribute. This latter indicates whether or not the relationship is an aggregate or reference. Note that this metamodel has similar concepts to those used to represent it, i.e. Ecore metamodel, but it is convenient differentiate between a metamodel (*Business Entity*) and the metamodeling language (meta-metamodel) used to create it (e.g. Ecore).

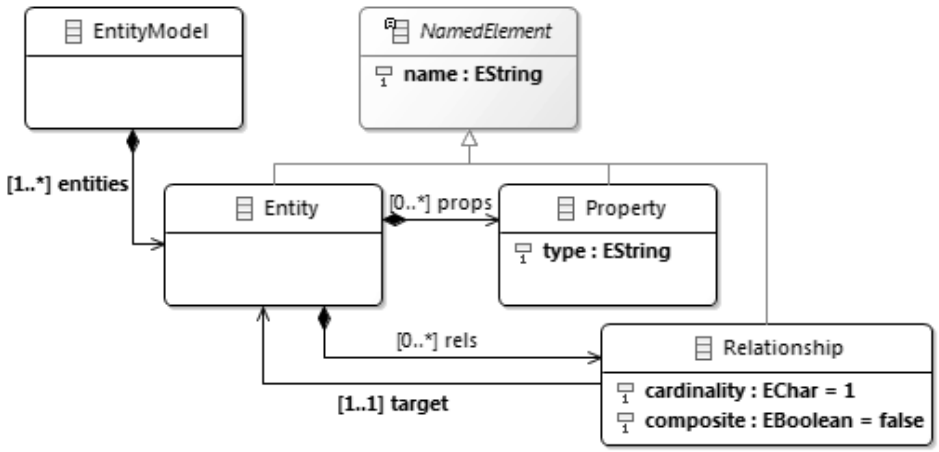

Figure 2.1: A simple example of metamodel: Entity metamodel.

A metamodel also includes a set of rules that impose restrictions on the models that can be instances, which cannot be expressed in the metamodeling language. These rules are commnly expressed in the *OCL* language [65] or languages inspired by it. For example, the following rule expressed that an entity model cannot have two entities with the same name:

```
invariant DifferentNames4Entities('Two entities cannot have the same name'):
        self.entities->forAll(e1,e2|e1 <> e2 implies e1.name <> e2.name);
invariant NotSymmetricAggregation:
        self.entities->forAll(e1,e2| e1.aggregates(e2) implies not e2.
            aggregates(e1));
```

In this thesis, all the devised metamodels have been defined with Ecore.

2.1.2 Domain-Specific Languages (DSLs)

Models are normally created from textual or graphic specifications In MDE, a DSL consists of three elements [61]: (i) its abstract syntax: it is expressed by means of a metamodel that defines the DSL concepts and the relationships between them; (ii) its concrete syntax: textual or graphical notation defined on the metamodel; and (iii) its semantics: it is normally a translational semantics expressed in form of a model transformation chain that establishes a mapping between the DSL and another language that has a well-defined semantics, e.g., general-purpose programming languages (GPL) as Java or Python.

Eclipse Modeling Project includes two popular DSL definition workbenches: Xtext [22] for textual DSLs and Sirius [107] for graphical DSLs. These tools automatically generate an model editor and injector. A model injector is a tool that parses DSL code or graphical diagrams to generate the corresponding model that conforms to the DSL metamodel.

In this thesis, the SkiQL language has been created by using Xtext.

2.1.3 Model transformations

An MDE solution consists of a chain of model transformations that generate software artefacts from one or more input models. There are three types of transformations: *model to model* (*M2M*), *model to text* (*M2T*) and *text to model* (*T2M*).

The *M2M* transformations generate a target model from a source model establishing a *mapping* between the elements defined in both metamodels. The transformations are usually expressed in declarative languages such as *ATL* [81] and *QVT* [64]. They can also be implemented with GPLs that access a model management API, e.g. EMF API [110].

The *M2T* transformations generate textual information (for example, source code or XML documents) from an input model, and they are usually the last step in the model transformation chains. They are usually expressed in template languages such as Acceleo [1] or template mechanisms provided by GPL as Xtend [9]. M2M transformations are often used as intermediate stages in the transformation chains to reduce the semantic gap between the input models and the artefacts that must be generated.

The case of *T2M* transformations are needed in software reengineering or reverse engineering scenarios, where initial models are obtained from text, normally GPL code. A text to model transformation process is commonly named "model injection", and it can be implemented in three forms: using a DSL definition workbench; GPL along with a model management API; and, rarely, using a language tailored to the kind of transformation as *Gra2MoL* [80].

In this thesis, M2M transformations have been written in Java and EMF API is used to manage Ecore metamodels. No M2M transformation languages were used mainly due to the complexity of the involved transformations, as discussed in [28]. All M2T transformations have been written with the template mechanism of Xtend.

2.2 The User Profiles Running Example

The "User Profiles" running example used along this thesis is shown in Figure 2.2. The figure shows the conceptual schema that will be used to build a database example for each paradigm integrated in our unified model. The extraction algorithms will be executed for that databases. The schema will also be used to validate the algorithms developed for the static code analysis, and to illustrate examples of SkiQL queries. In this chapter, this schema example will allow illustrating each presented data model.

"User Profiles" schema could be an excerpt of the conceptual schema of a movie streaming platform, which is expressed as a *UML* class model. It has 3 entities labeled *Movie*, *User*, and *Address*, and 3 relationships: a user aggregates an *address*, a user has zero or more *favorite movies*, and a user has zero or more *watched movies*. *User* has the attributes *name*, *surname*, and *email*; *Address* has *city*, *street*, *number*, and *postcode*; and *Movie* has *title*, *year*, and *genre*.

When instantiating each database, we will suppose that there are 2 variations for the *Address* entity type: {*street*, *number*, *city*}, and {*street*, *number*, *city*, *postcode*}; and 2 variations for *User* that vary in the relationships: either *favoriteMovies* and *watchedMovies* coexist, or only *watchedMovies* is present, and in the attributes: the *surName* attribute is only present when both relationships are.

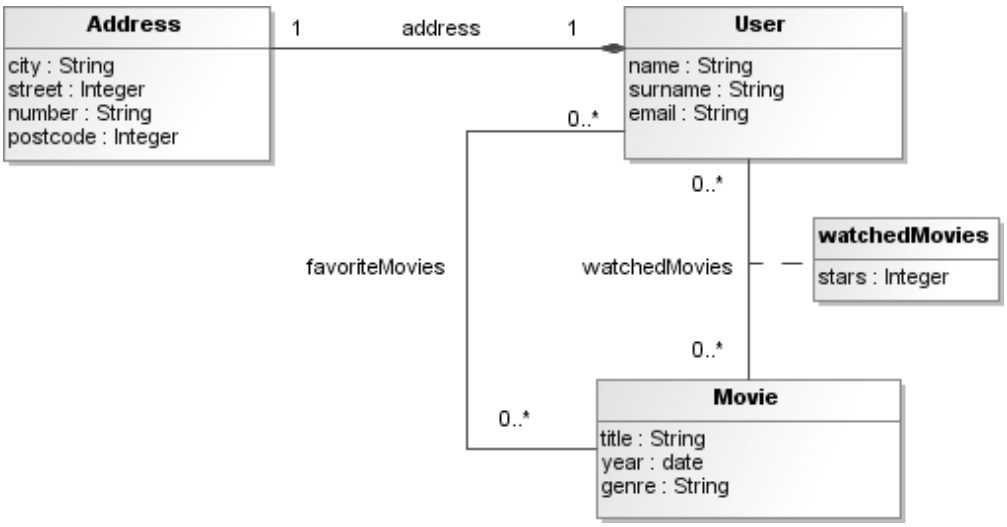

Figure 2.2: "User profile" running example schema.

2.3 Data Modeling

A data model provides a set of concepts to specify the structure and constraints of a database type, and a *schema* results of applying a concrete data model on a domain or problem. A schema is therefore an instance of a data model. Given a particular data model, textual and graphical languages can be defined to express schemas.

Data models (and therefore schemas) can be defined at different levels of abstraction. Typically, they are classified in three categories: conceptual, logical, and physical. *Conceptual* schemas represent the domain of an application in a platform-independent way. *Logical* schemas describe data structures and constraints, but providing physical independence. Finally, *Physical* schemas include all details needed to implement a logical schema on a specific database system.

At the logical or physical level, a *unified* or *generic data model* can be defined to integrate concepts from several data models with the purpose of offering a uniform representation. When using a unified model for n data models, instead of managing $n \times (n-1)$ mappings (each data model with the others), only $n + n$ mappings are needed (between the unified and each of the integrated data models in both directions.)

Relational data model is the most common data model used, which represents the data in form of tables that are organized in columns. NoSQL database systems are classified in sev-

eral data models, but commonly the term NoSQL refers to four of them: *columnar*, *key-value*, *document*, and *graph* systems. In the following sections we will describe each NoSQL data model as well as the relational model. Unlike relational model, there is not a specification or standard for NoSQL models, and systems classified in the same paradigm have differences in the form of structuring data. Therefore, the data models here presented reflect our understanding of the logical structure for the most popular system of each paradigm.

2.3.1 The Graph Data Model

In graph systems (e.g., *Neo4j* and *OrientDB*), a database is organized as a graph whose nodes (a.k.a. vertex) and edges (a.k.a. arcs) are data items that correspond to database entities and relationships between them, respectively. Edges are directed from an *origin node* to a *destination node*, and more than one edge can exist for the same pair of nodes. Both nodes and edges can have *labels* and *properties*. Labels denote the entity or relationship type to which nodes or relationships belong, and properties are key-value pairs. This is the so called *labeled property graph data model* [12], that most NoSQL graph systems implement.

Graph databases are commonly schemaless, so there may exist nodes and relationships with the same label but different set of properties. Moreover, the same label can be used to name relationships that differ in the type of the origin and/or destination nodes. Thus, graph databases can have structural variations as explained in Chapter 4.

For this kind of graph store, more specifically Neo4j graph stores, we have abstracted the following notion of logical *graph data model*, which is represented in form of *UML* class diagram in Figure 2.3:

i) A graph schema has a name (that of the database) and is formed by a set of *entity types* and a set of *relationship types*.

ii) An entity type denotes the set of nodes with the same label (or set of labels).

iii) Entity types can be single-label or multi-label depending on whether they have one or more labels.

iv) A relationship type denotes the set of relationships with the same label (or set of labels). A relationship type has origin and destination entity types.

v) Entity and relationship types can have structural variations.

vi) A structural variation is characterized by a set of properties that is shared by elements with the same set of labels.

vii) A property is a pair that mimics the property of a node or relationship in the graph, having a key and the scalar data type that corresponds to the values of the property.

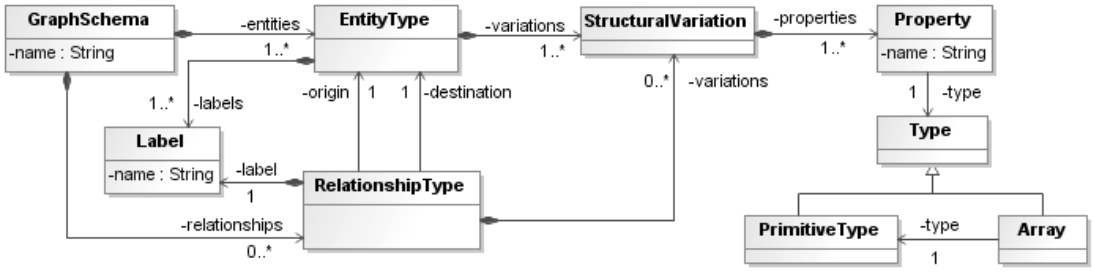

Figure 2.3: Graph Data Model.

Figure 2.4 shows a graph database for the "User Profiles" running example. It has three entity types labeled *Movie*, *User*, and *Address*, and three relationship types labeled *FAVORITE_MOVIES*, *WATCHED_MOVIES*, and *ADDRESS*. In the figure, nodes are represented as circles, and relationships as arrows. Nodes having the same labels (i.e. entity type) are filled with the same color. In this example, gray for *Address*, white for *User*, and black for *Movie*. Nodes only show a property for each entity type: *title* for *Movie*, *name* for *User*, and *street* for *Address*. Relationships are tagged with their relationship types, and no properties are shown. We suppose that there are the variations indicated in Section 2.2.

2.3.2 The Document Data Model

Document databases (e.g., MongoDB and Couchbase) are organized in collections of data recorded for a particular database entity (e.g., *Movie*, *User*, and *Address* in the running example). Data are stored in the form of semi-structured objects or documents [10, 27] that consist of a tuple of key-value pairs (a.k.a. fields). Keys denote properties or attributes of the entity, and the values can be atomic data (e.g. Number, String, or Boolean), nested or

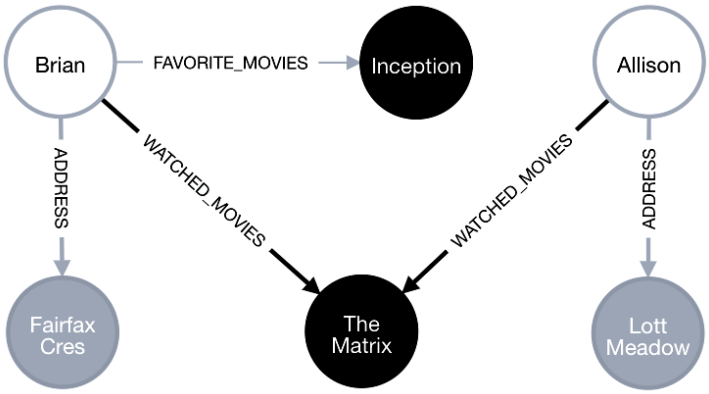

Figure 2.4: User Profiles Graph Database Example.

embedded documents, or an array of values. Also, a string or integer value can act as a reference to another document, similar to foreign keys in relational systems, although usually no support for consistency is provided.

Semi-structured data is characterized by having its schema implicit in itself [27]. Thus, document databases are commonly schemaless, and a collection can store different *variations* of the entity documents. Usually, document databases maintain data in some JSON-like format.

For document databases, more specifically MongoDB stores, we have abstracted the following notion of *document data model*, which is represented in form of a *UML* class diagram in Figure 2.5:

i) A document schema has a name (that of the database) and is formed by a set of *entity types*.

ii) An entity type denotes a collection of documents stored in the database.

iii) Entity types have one or more structural variations.

iv) A structural variation is characterized by a set of properties that are shared by documents of the same collection.

v) Properties have a name and a type, and can be attributes, aggregates, or references.

vi) Attributes denote object's fields whose value is of scalar or array type. An attribute is specified by the name of the field and the type of its value. We suppose that there exists an attribute that acts as the key of the Entity type (e.g., "`_id`" in MongoDB).

vii) Aggregates denote object's fields whose value is an embedded object. An aggregate is specified by the name of the field and the variation schema of the embedded object.

viii) References denote object's fields whose values are references. A reference is specified by the name of the field and the type of its value.

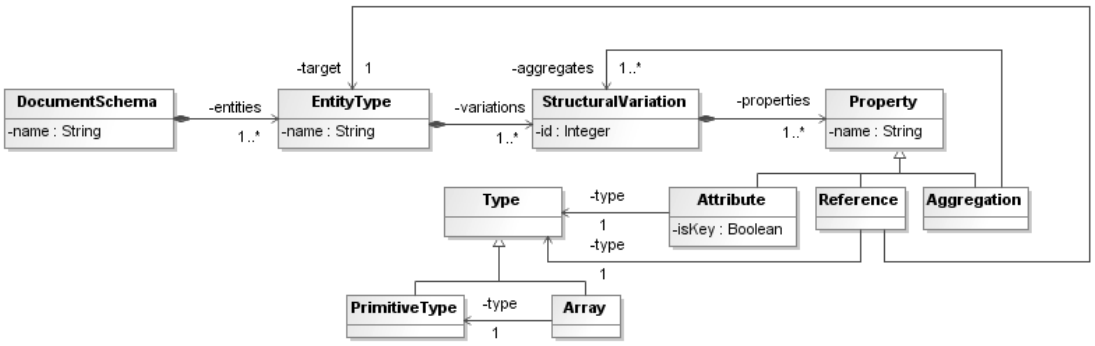

Figure 2.5: Document Data Model.

Figure 2.6 shows how the "User Profiles" running example would be stored in a document database. Instead of using JSON notation, we depicted the database objects in a representation that remarks their nested structure and the references between objects. There are two collections: *User* and *Movie* objects, and the relationships are as follows. *User* objects aggregate *watchedMovies* objects with two properties: the *stars* attribute and the *movie_id* reference that records the *id* value of a movie object (arrow from *movie_id* to *Movie* objects); *watchedMovies* objects are recorded in an array. To record favorite movies, *User* has the *favoriteMovies* array of references to *Movie* objects. The user addresses are stored as an *address* aggregate object of users. While graph databases rely on references (i.e. relationships in graph store terminology) to connect data items, and aggregation is normally not available to compose data, the opposite is true in document database systems.

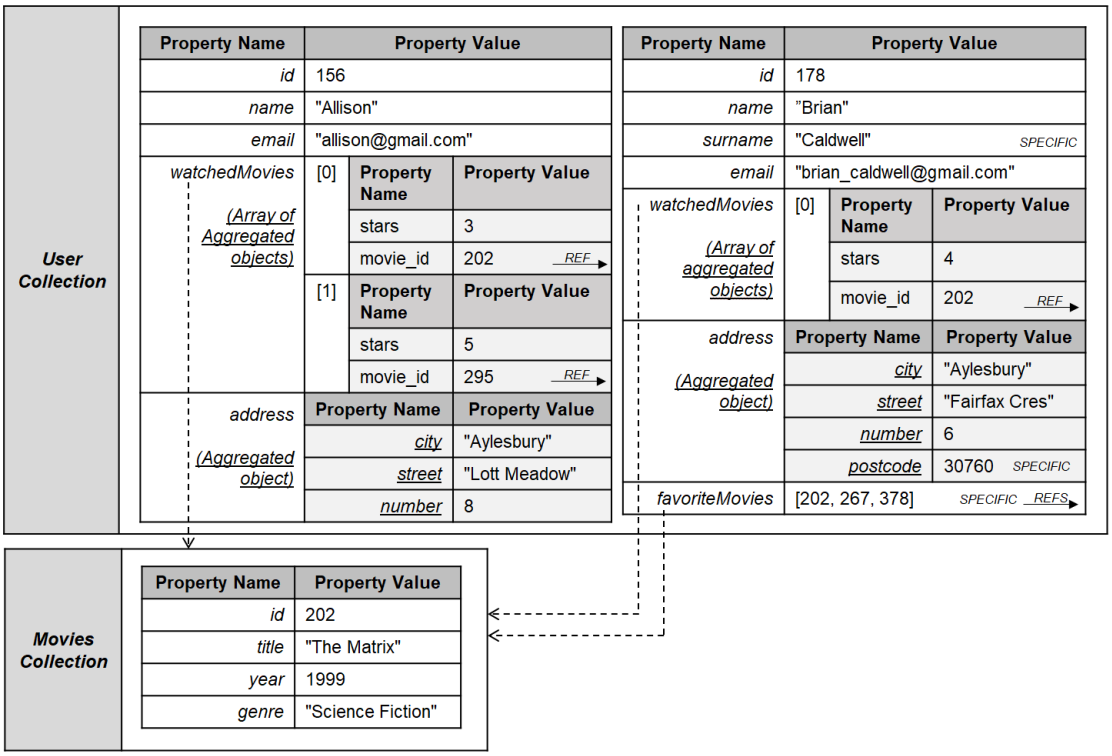

Figure 2.6: "User Profiles" Document Database Example.

2.3.3 The Key-Value Data Model

Key-Value (K/V) stores conform to the simplest physical data model of NoSQL systems. A K/V store is an associative array, dictionary, map, or *keyspace*, that holds a set of key-value pairs, usually lexicographically ordered by key. As such, they are used to record data with a simple structure, and references and aggregations are not primitive constructs to build up data. They usually store a single entity type (e.g. user profile, user login, or a shopping cart), although data of several entity types could co-exist in the same keyspace.

Like document and columnar systems, K/V stores can record semi-structured objects. Several techniques can be used to encode a tree-like structure into key-value pairs, which use normally *namespaces* to build hierarchical key values. We chose one of the most commonly used encoding patterns * to which we will call the *flattened key pattern of compound objects* or simply *flattened object-key pattern*.

*Encoding pattern, Redis website: https://redis.com/redis-best-practices/data-storage-patterns/object-hash-storage/.

When using this pattern, the key of every pair not only acts as the identifier of the object, but also encodes the name of a property of the entity type, in a similar format to XPath or JSONPath [60]. Keys are built with a separator to differentiate between the object identifier and the property name (e.g., a colon: "`<id>:<property>`"). It can also be used to differentiate the entity type if different namespaces are not used (e.g., "`<entity-type>:<id>:<property>`"). When a property aggregates an object, it is possible to use another separator to express properties of the aggregated object (e.g., a dot: "`<id>:<property>.<aggregated-property>`"), or an index to represent objects of an array (e.g., "`<id>:<property>[<index>]`"), that can have properties (e.g., "`<id>:<property>[<index>].<aggregated-property>`"). Figure 2.7 shows an K/V database example that illustrates the usage of this encoding for the "User Profile" running example. Using this pattern, a database object consists of several entries in the database, all of them sharing the same object identifier. Note that the order of the separated elements of the key may vary depending on the specific queries needed by the application, as the keys are lexicographically ordered.

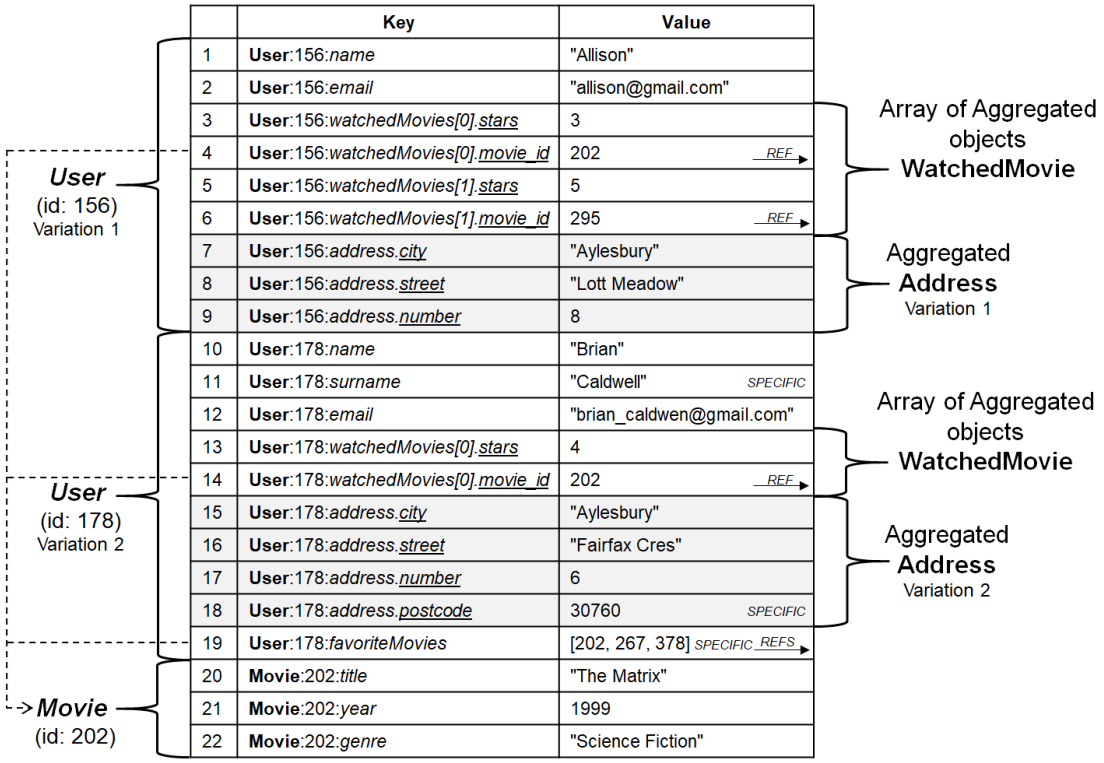

Figure 2.7: Key-Value Database Example for running example.

K/V systems are schemaless, and several structural variations of an entity type can therefore exist in the database. In Figure 2.7, the variations of the running example can be observed.

Taking into account the use of the *flattened object-key pattern*, the document data model presented in Section 2.3.2, and shown in Figure 2.5, can also be used for K/V systems by modifying the *Key* notion. In this case, every database object also has a key, but it is not associated to any attribute. A namespace would correspond to an entity type or either, if only one namespace is used, each different entity type will have a different "`<entity-type>`" key prefix. We will use the term *aggregate-oriented data model* to group the Document, Key-Value, and Columnar data models, as suggested in [98], because they include the same concepts in their respective data models.

A set of data types are available for keys and values, which vary on each system. Keys are normally stored as byte-arrays or strings, which can follow formats as those indicated above. Regarding the data types of values, they usually provide basic scalar types as well as common collection types.

2.3.4 The Columnar Data Model

In columnar databases, data is structured in a similar way to relational databases. In the most popular columnar databases (Hbase [73] and Cassandra [30]),[†] a *database* or *Keyspace* schema S is composed of a set of tables $T = \{t_i\}, i = 1..n$, and each table t_i usually stores data of a single entity type. As in relational databases, each table has a name, and is organized in rows and columns, but rows have a more complex structure than in relational tables because they are organized in column families. A table t is therefore defined in terms of a set of *column families* $F^t = \{F^t_j\}, j = 1..m$. Moreover, each row r belonging to a table t contains a *row key*. Figure 2.8 shows an example of columnar database for the running example, which has the *User* and *Movie* tables. The *User* table contains three column families: *User*, *Address*, and *WatchedMovies*. The *Address* and *WatchedMovies* relationships of the running example are represented as column families, and the *FavoriteMovies* relationship is represented as a column of the *User* family, which records an array of references to *Movie*. In the case of

[†]This can be observed in `https://db-engines.com/en/ranking`. Cassandra appears in the 10nd position and HBase in the 22nd position as of March, 2021.

Cassandra, column families will be equivalent to *User Defined Types* (UDTs): in a Cassandra table, the type of an attribute can be either a predefined type or a UDT. Thus, the *User* table could have the four attributes: *name* and *email* whose type would be *Text*, and *address* and *watchedMovies* whose types would be the UDTs *Address* and *Movie*, respectively.

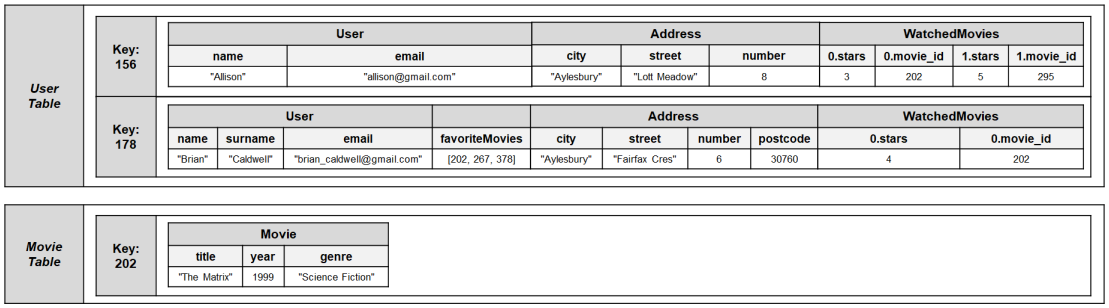

Figure 2.8: Columnar database example for the running example.

Columnar databases also record semi-structured data, and they are normally schemaless, which means that structural variation is possible: the set of columns present for each column family can vary in different rows. In the Figure 2.8, the structure of the *Address* object is different for each of the two *User* objects; moreover, the second row has an additional *surname* column for the *User* column family.

We will suppose that a table has a *default* column family that includes the attributes of the root entity type that corresponds to the table. The rest of column families represent aggregated entity types. (Again, in the case of Cassandra, the set of attributes in the table that are not UDTs will form the default column family.) In the example, the default column family is *User*, with *Address* and *WatchedMovies* as aggregated entities. Note that *WatchedMovies* aggregates an array of objects, so the name of the columns is formed by using the *flattened object-key pattern*[‡] ("`<property>.<index>.<aggregated-property>`"), where the property name is the name of the column family and can be omitted. For example: "`0.stars`", "`0.movie_id`" in Figure 2.8.

As column families are considered a way of embedding objects into a root object, the data model defined for Key-Value and Document stores is applicable for columnar stores, that is, the *aggregate-oriented data model*.

[‡]Flattened object-key pattern, HBase website: https://hbase.apache.org/book.html#schema.casestudies.custorder.obj.denorm.

2.3.5 The Relational Data Model

Unlike NoSQL logical data models, there exists a standard relational data model which is formally defined through relational algebra and calculus. Being "schema-on-write" is another significant feature that differentiates relational databases from NoSQL stores: schemas must be declared prior to store data in tables. The relational model is based on the mathematical concept of *relation* and its representation in form of *tables* [37]. A detailed description of the relational model can be found in [113, 47].

A relational schema consists of a set of relation schemas. Each relation schema specifies the relation name, the attribute names and the domain (i.e., type) of each attribute. Relationships between relations are implicitly represented by key propagation from a relation schema to another (one-to-one and one-to-may relationships) or either by a separated relation schema (many-to-many relationships). Therefore, relation schemas can represent entity types or relationship types. A relational schema is instantiated by adding tuples to each relation. Each relation has one or more attributes that form the key (*primary key*), and each tuple is uniquely identified by the values of the key attributes. Relations are represented as *tables*, and the term *column* is used to refer to the attributes, while *rows* name the tuples of a relation. A table can declare *foreign keys*: one or more columns that reference to the primary key of another table in a key propagation.

User

user_id	*name*	*surname*	*email*	*city*	*number*	*street*	*post_code*
156	"Allison"	null	"allison@gmail.com"	"Aylesbury"	8	"Los Santos"	null
178	"Brian"	"Caldwell"	"brian_caldwell@gmail.com"	"Aylesbury"	6	"Fairfax Cres"	30760

FavoriteMovies

user_id	*movie_id*
178	202
178	267
178	378

Movie

movie_id	*title*	*year*	*genre*
202	"The Matrix"	1999	"Science Fiction"

WatchedMovies

user_id	*movie_id*	*stars*
156	202	3
156	295	5
178	202	4

Figure 2.9: "User Profile" relational example.

Figure 2.9 shows a relational database example for the schema of the running example. *User* and *Movie* tables represent the entity types of identical name, *WatchedMovies* and *FavoriteMovies* tables represent the many-to-many relationships from *User* and *Movie* in the conceptual schema of the running example, and *User* aggregates *Address* by incorporating

its attributes. Note that *Address* could be a separate table related by foreign key, but it has been integrated into *User* because they hold a one-to-one relationship.

In the last four decades, conceptual and logical schemas for relational systems have been extensively studied, and a lot of methods and tools are available for using them in the whole database life cycle. Entity-Relationship (ER) [113], Extended ER (EER) [47] and Object-Orientation modeling are the most widely used formalisms to model conceptual and logical schemas for relational databases.

3

State of the Art

In our research work, we have tackled four problems in the Data Engineering area: (i) definition of a unified metamodel for NoSQL and relational databases, (ii) design and implementation of schema extractors from NoSQL data and database code, (iii) development of a schema query language, and (iv) automate a NoSQL database refactoring. In this chapter, we have therefore organized the study of the related work in the following sections. First, other unified or generic metamodels will be presented. Next, we have separated the study of schema extraction into two sections, one dedicated to extraction from data and another from database application code. In this latter, we have also included Code metamodels published so far. In the fourth section, schema query language proposal will be discussed. and we will finally detail some data modeling tools. At the end of each section, works will be contrasted to our proposals.

3.1 Generic Schema Metamodels

The definition of generic or unified data models to integrate different data models is not a novelty, but that this idea has been applied from early years of nineties, for instance unified models were considered to develop heterogeneous distributed database systems, as indicated in [99], and build generic database engineering environments as GER [77, 72].

With the predominance of relational databases, the interest by them disappeared. Now, unified models are resurfacing because to the adoption of NoSQL stores. For example, Amazon has defined PartiQL [2] to access its variety of storage systems, and the Hackolade data modeling tool recently announced the creation of a generic model to represent all the data models that it supports [69]. For the ErWin data modeling tool, Allen Wang also proposed a unified model in 2015 [118], but its architecture has not been implemeted yet, as far as we know. Also, Atzeni et al. have defined a unified model to access NoSQL stores in a platform-independent way [16, 14]. In the area of model management, unified models have also been proposed to uniformly represent different data formats, and are remarkable the proposal of Atzeni et al. [17], and GeRoMe metamodel [83]. In this section, we will analyze all these proposals of unified models.

3.1.1 Generic Entity/Relationship (GER)

DB-Main was a long-term project initially aimed at tackling the problems related to database evolution [77, 72]. For this, the creators will consider to develop a platform able to support the main data models, namely relational, object-oriented, object-relational, and earlier models as network and hierarchical. The DB-Main approach was based on three main elements: (i) The Generic Entity/Relationship (GER) metamodel to achieve platform- independence; (ii) A transformational approach to implement operations such as reverse and forward engineering, and schema mappings; and (iii) A history list to record the schema changes [77].

Here, our interest is focused on the two former elements. The GER generic metamodel was defined as an extension of the ER metamodel [113]. Conceptual, logical, and physical models could be represented in GER. Models for a particular paradigm, system, or methodology were obtained by means of (i) selecting necessary GER elements, (ii) defining structural predicates to establish legal assemblies of that elements, and (iii) choosing an appropriate visual diagram. Regarding schema transformations, a set of basic transformations were defined, and the signature of each of them (name, input, and output) was specified in a particular format to be used to record changes in the history list. When obtaining a schema, it is necessary to define a sub-model for a specific database system and develop an extractor for each database system.

GER metamodel is the EER metamodel [59] extended with (i) *Entity domains*, (ii) *Value domains*, (iii) *Entity Relation schemata*, iv) *Relationship Relation schemata*, and (v) *Constraints*. Figure 3.1 shows the five new elements and the connections between them in order to facilitate the understanding of these elements that are defined below.

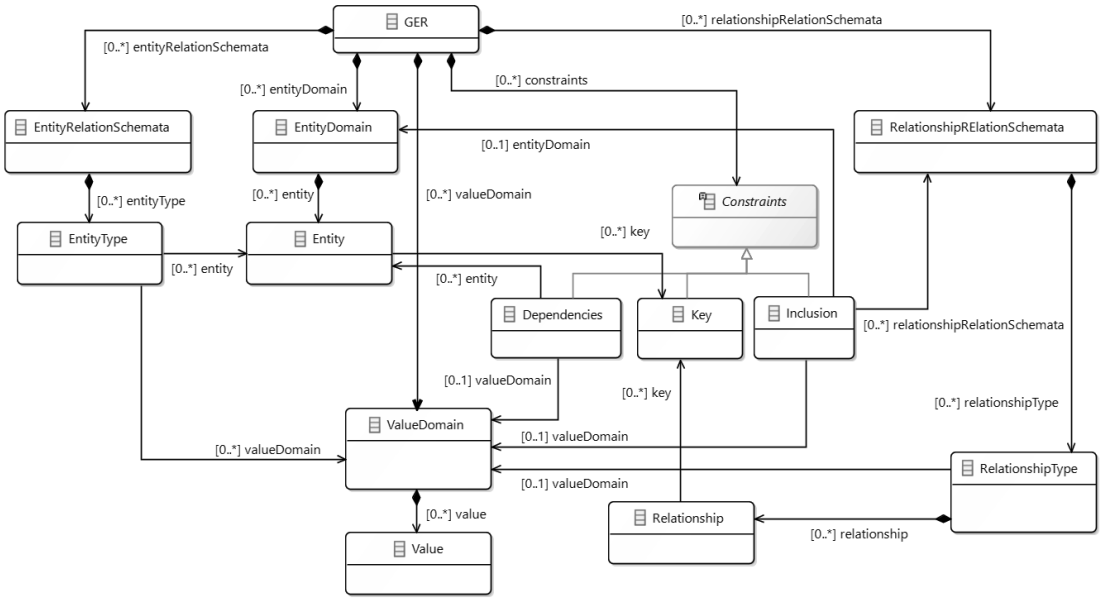

Figure 3.1: The new aspects of the Generic Entity/Relation (GER) metamodel.

An *Entity* belong to one *Entity Domain* and each *Entity Domain* can belong to another *Entity Domain*, called *superdomain*. An *entity domain* can be created by performing set operations like union, intersection or difference of other entity domains. All entities of an *entity domain* conform to an entity type that results of the union of all the *common* properties of all entities that belong to the entity domain.

A *value domain* refers to the set of values (i.e., a piece of data) of a simple basic or complex type. A simple type has atomic values as real, integers or strings, and a complex type is formed has several simple or complex types. Thus, a complex value is any element or elements of the constituent value domain or the Cartesian product or powerset of them.

An *entity relation schemata* establishes the connection between different attributes of different entities and it is defined as the primary key of the relation schema. A relationship is a set of connected-entities with a set of values, and it conforms to a relationship type. Relationship types are constructed with all the *common* properties of the relationships to

which the type belongs. For this, a *relationship relation schemata* defines the attributes of the relationship types. Relationship constraints such as cardinality are supported.

There are three types of *constraints*: *Keys*, *Dependencies* and *Inclusion*. All entities and relationships have one or more *key*s represented with attributes. A *Dependency* is the union of attributes of different entities, and an *Inclusion* is the connection between two entity sets (as entity domains), value sets (attributes) or relations (relationship relation schemata).

Over the years, DB-Main becomes a powerful data engineering environment that can currently be acquired from the Rever company.*

3.1.2 Generic metamodels for Model Management

Model Management (MM) is an approach aimed to solve *data programmability* problems which normally involve complex mappings between data schemas of different sources [21, 20]. A set of operators between models are proposed, such as *match*, *union*, *merge*, *diff*, or the *modelgen* operator that generates a schema from another. In [21], building a universal metamodel is considered a feasible way of developing tools to specify mappings, although it does not seems the more adequate alternative because of the large complexity of the required metamodel. In this section, we will present two universal metamodels created for applying model management [17] and [83].

Paolo Atzeni et al. defined a *universal metamodel* based on a three-level architecture similar to those defined in the EMF framework: a metamodeling language is used to define metamodels, which, in turn, are used to create models (schemas in our case) [17].

In [17], a set of 13 meta-constructs were defined to represent the concepts used in different data formalisms as can be seen in the table 3.1 and are shown in Figure 3.2 (extracted from [17]).

This proposal overlooked the already existing MDE frameworks, in particular EMF/Ecore. Instead, the authors started from scratch, and they even proposed a dictionary structure to store models as instances of the universal metamodel. Schemas are expressed by indicating, for each element, the construct at the level of the data model from which is instantiated, and for each of these constructs its meta-construct at the level of the universal metamodel. They offer the possibility of mapping the following data schema: Entity-Relationship, Bi-

*DB-Main website: https://www.dataengineers.eu/en/db-main/.

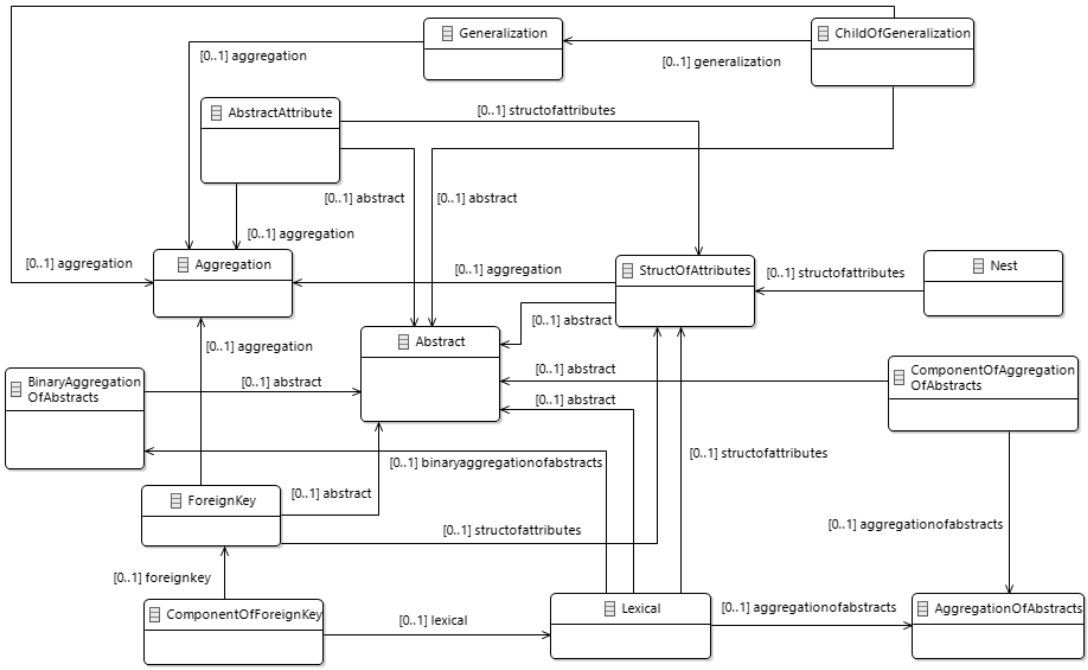

Figure 3.2: The Meta-Construct of the Supermodel of Atzeni et al. (Extracted from [17]).

nary Entity-Relationship, Objects (UML class diagram), Object-Relational, Relational and XSD. The metamodel was accompanied by a basic tooling for textual and graphical visualization.

In the case of *GeRoMe* [83], *role-based modeling* was applied to define a metamodel able to represent different data models to allow manipulating them. To define the GeRoMe metamodel, EER, Relational, OWL-DL, XML Schema, and UML were analyzed with the aim to identify their similarities and differences. Then, a set of roles was established, and the role-based metamodel created.

An element of a GeRoMe model is an empty object without any characteristic, as characteristics are included in roles in form of properties. Whenever a role is assigned to an object, this object will be able to play that role, i.e., the role properties (attributes and methods) are added. An object can play different roles. The existing roles cannot be modified, the objects can only be modified as they can change the role played at any time. Role inheritance is supported.

Figure 3.3 shows the GeRoMe metamodel as exposed in [83]. All the roles and the con-

Meta-construct	Definition
Abstract	Represents concepts such as ER entities or OO classes.
Aggregation	Set of elements with heterogeneous structure, such a table in relational schemas.
StructOfAttributes	An abstract or aggregate element, and it can also recursively aggregate other StructOfAttributes.
AbstractAttribute	A reference from an abstract, an aggregation, or a StructOfAttributes to an abstract.
Generalization	Indicates that an abstract is the parent element of an inheritance hierarchy between two abstracts.
ChildOfGeneralization	Similar to *Generalization*, it indicates that an abstract is the child in an inheritance hierarchy between two abstracts.
Nest	Used to join different StructOfAttributes.
Lexical	Any value attribute with type. Relational columns are mapped to this element.
ForeignKey	A constraint of relation between Abstract, Aggregation or StructOfAttributes. Foreign keys of relational schemas are an example.
ComponentOfForeignKey	Similar to the previous one, specifies the lexicals involved in a relationship, for example columns that conform to a foreign key in a relational schema.
BinaryAggregation OfAbstracts	Denotes that two abstracts are related in binary form.
AggregationOfAbstracts	Denotes an aggregate relation of two o more abstracts.
ComponentOfAggregation OfAbstracts	Denotes that an abstract belongs to an *aggregationOfAbstracts*.

Table 3.1: Set of 13 Meta-constructs defined in [17].

nections between them are shown in the figure. GeRoMe specifies a total of 48 roles that are classified into three categories: *structural elements*, *derivations*, and *constraints*. The most important roles are described in the table 3.2.

In mid-nineties, role-based modeling approaches received attention in the context of object-oriented programming to model the multiple-classification and object collaborations [100]. However, that interest has decreased over the years because languages and tools do not support the notion of *role*.

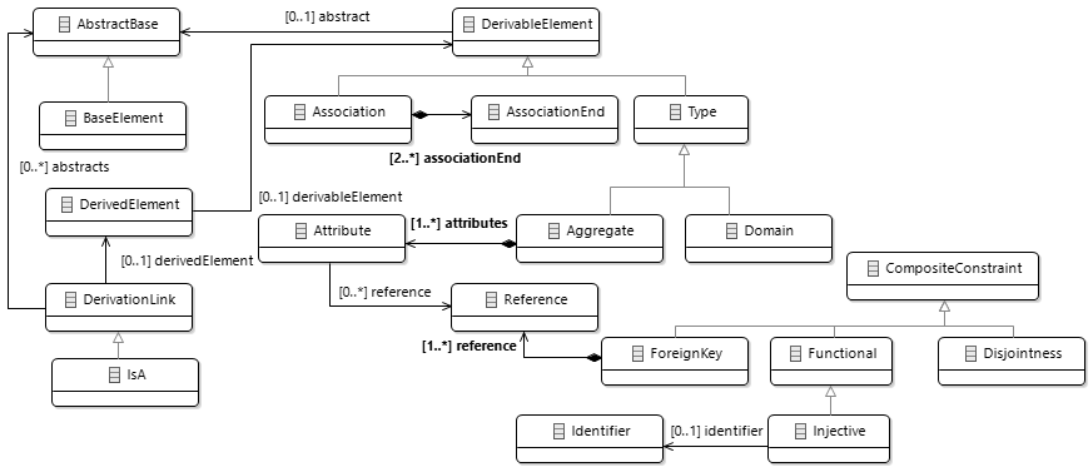

Figure 3.3: The GeRoMe metamodel (Extracted from [83]).

3.1.3 The NoSQL abstract model

More recently, several metamodels have been proposed to represent NoSQL generic data models. *SOS* is a metamodel designed to represent schemas of aggregate-based stores [15], with the purpose of achieving a uniform accessing. SOS is shown in Figure 3.4 (taken from [15]), where a NoSQL schema consists of a set of collections (*Set* metaclass), which can contain *Struct*s and *Attribute*s. An *Attribute* represents a key-value property, and a group of key-value pairs is modeled as a *Struct*. *Struct* and *Set* can be nested.

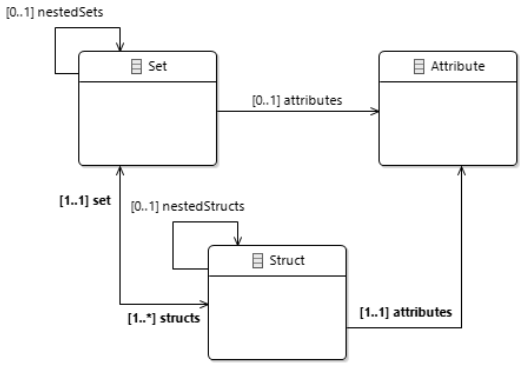

Figure 3.4: The SOS Metalayer of Atzeni et al. (Taken from [15]).

Later, SOS evolved to the *NoAM* (NoSQL Abstract Model) metamodel [14], which was defined as part of a design method for aggregate-based NoSQL databases [16]. NoAM was

Structural roles of GeRoMe	
Aggregate	An element with structure that contains a set of attributes. It can model concepts as entity type, relationships type or class.
Attribute	Subrole of Property (which in turn is a subrole of Particle) defines a data with cardinality constraints (inherited from particle).
Type	Superclass that defines the type system of the metamodel, subtypes are primitive types (Domain) or Aggregate.
Domain	Predefined basic simple types such integer or string.
Association	A relationship between objects.
AssociationEnd	The properties of associations.
***Derivations* roles of GeRoMe**	
BaseElement	Element to be extended.
DerivedElement	Element created as result of other existing BaseElements.
DerivationLink	Connect different BaseElements to a DerivedElement.
IsA	Define a specialization relationship. If used connected to other DerivedElements defines a new type with their properties.
***Constraints* roles of GeRoMe**	
Disjointness	To define two or more types to be joint.
Functional	Declares a property as a function.
Identifier	Names an object.
Injective	Declares a relationship between two elements. If along with Identifier role acts like a primary key.
ForeignKey	A set of references defines a reference to a object with role Identifier.

Table 3.2: The most important roles of the GeRoMe metamodel [83].

designed as an intermediate representation to transform aggregate objects of database applications into NoSQL data. A NoAM database, as can be seen in figure 3.5, is a set of collections that contains a set of blocks. A block contains a set of key-value pairs, and each block is uniquely identified by a key. In [16], several strategies are described to represent a collection of aggregate objects in form of a NoAM database.

3.1.4 ERwin Unified Metamodel

ERwin Unified Data Modeler (ModelSet) is a project outlined by Allen Wang in an article published in infoQ [118]. The aim of this project is very close to ours, however, to our

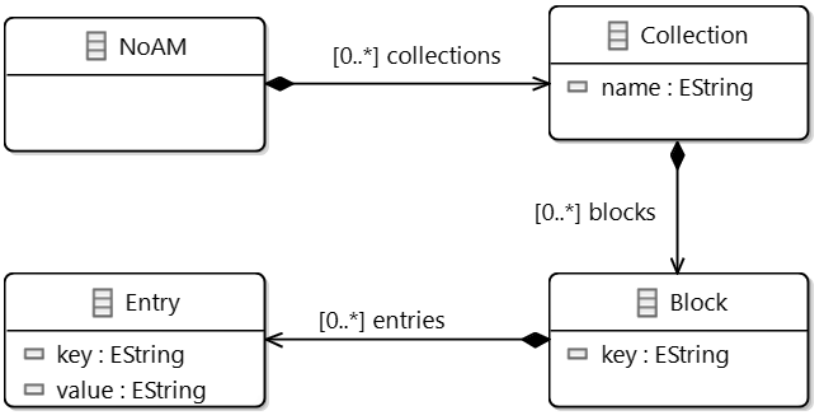

Figure 3.5: The NoSQL Abstract Model (NoAM) metamodel.

knowledge, no results have not been published yet, and ErWin has not developed ModelSet. Wang proposed a simple unified logical model to integrate three kind of schemas: columnar, document, and relational. The logical/conceptual metamodel only includes four elements that refers to the schema representations.

The Figure 3.6 shows the main modeling constructs of the metamodel for schema definition: (i) *Entity* is used represent tables in relational stores, collections in documents stores and columns families in columnar stores. (ii) *Relationship* models different references between entities, this is used to represent foreign keys in relational stores. (iii) *Attribute* (of entities only) represent the minimal object of data such as columns in relational and columnar or key-value of a document in documents stores,and (iv) *Tags* used to extends other elements.

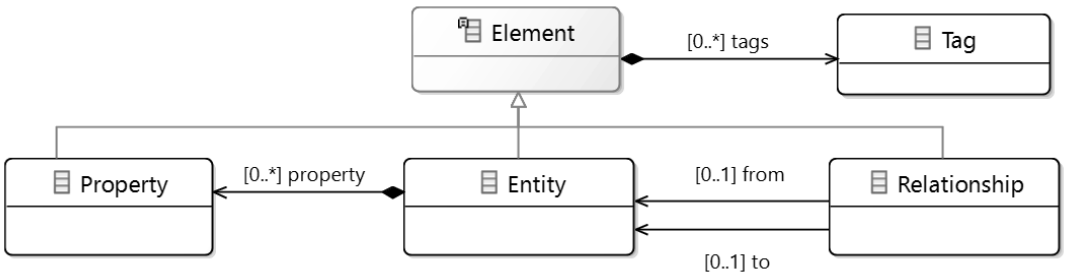

Figure 3.6: The ERwin Logical/Conceptual Metamodel (Taken from [118]).

The proposal also considered the creation of a physical model for each database system.

Query and data production patterns are defined on the logical model for its transformation into physical model. Several notations to represent schemas were proposed such as ER-Diagram, IE Notation, Document base Notation, and Column Family Notation. Wang indicated that the tool based on the unified model should support forward and reverse engineering. However, the Wang's article did not provide any detail related to the implementation, as the use of machine learning to infer schemas.

3.1.5 Typhon Modeling Language Metamodel

The Typhon project[†] is an European project aimed to create a methodology and tooling to design and develop solutions for polystore database systems. As part of this project, the TyphonML [114] language has been built, which allows schemas to be defined in a database system-independent way. Columnar, document, key-value, graph, and relational schemas can be defined with TyphonML. Typhon schemas can also express mappings from schemas to the physical representation.

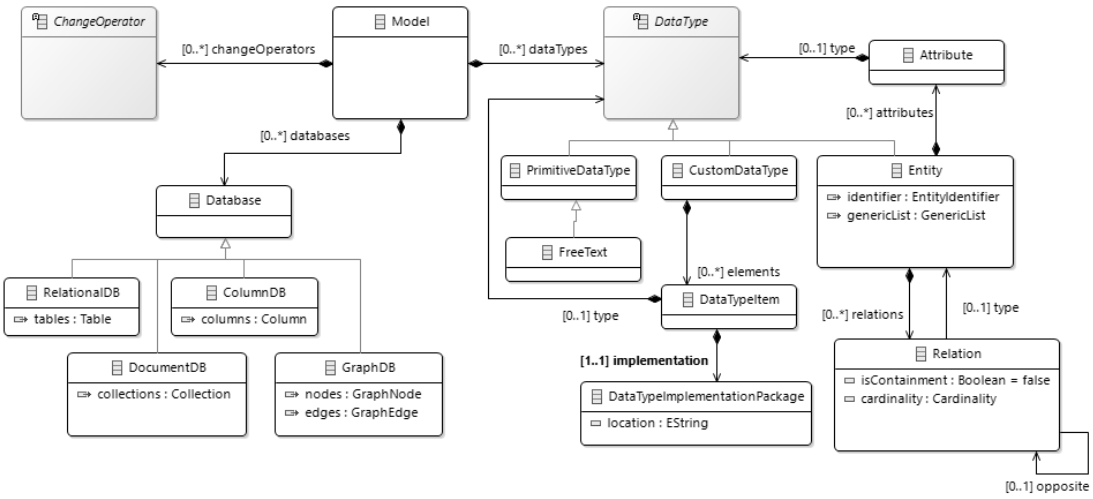

Figure 3.7: The TyphonML Metamodel (Taken from [114]).

Figure 3.7 shows the TyphonML metamodel, whose root element is *Database*. This class that represents the kind of database (RelationalDB, DocumentDB, ColumnDB or GraphDB). A *DataType* class model the data type that can be primitive types (*PrimitiveDataType*, text

[†]Typhon project website: https://www.typhon-project.org/.

FreeText), a user-defined types (*CustomDataType*) or database entities *Entity*. An *entity* is formed by a set of *Attribute*s and *Relation*s. *Attribute*s are a name and data type pair, and embedded objects can be represented: attributes whose type is another *Entity* type. References and aggregate are distinguished using the property *type*. Structural variation is not considered. To inspect TyphonML, we have observed that the support of graph is not completed, e.g. relationhip types are not represented.

3.1.6 PartiQL

Given the widespread usage of different data models, developers and companies face the problem of managing several query languages. Therefore, there exists a great interest in creating a universal query language for the variety of data managed in modern applications, and some proposals have recently appeared. Among them, the most relevant is the PartiQL model.

PartiQL [2, 7] is a query language created in Amazon to achieve independence of format and data store in accessing the variety of data stored (NoSQL, relational, and data lakes) used in the company. To achieve format and data store independence, PartiQL has been built on a generic data model able of representing tabular, nested, and semi-structured data, but not graph data. Moreover, it works both with schema-on-write and schema-on-read database engines. The parser of PartiQL is open-source and provides a reference implementation architecture to companies interested in its implementation [2].

PartiQL is backwards compatible with SQL-92, and its data model is formed by a set of types that represent the following kinds of values: (i) Scalar data whose type is defined with the Ion's type system [79]; (ii) Absent types in order to represent nulls and not present values; (iii) Arrays the elements of arrays and bags can be heterogeneous; (iii) Complex values can be formed by composing arrays, bags and tuples. Tuples are composed of pairs of key-values; (iv) Null-valued attributes are distinguished of missing attributes.

3.1.7 Data Modeling Tools

With the emergence of NoSQL systems, multi-paradigm data modeling commercial tools have proliferated. In our study of some of the most popular of these tools, we have found no evidence showing the use of a unified metamodel. Next, we contrast features of these

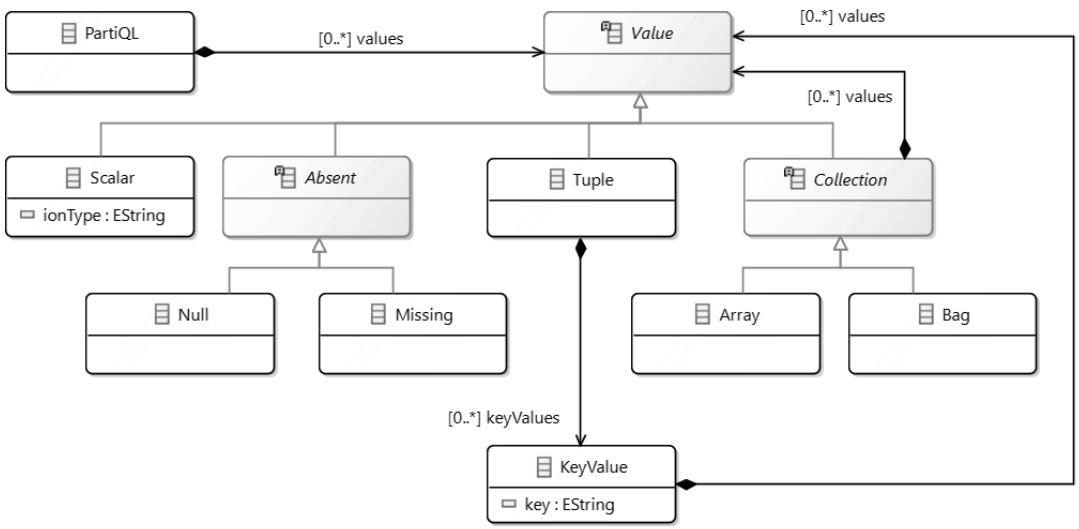

Figure 3.8: The PartiQL data model.

tools with those considered in our U-Schema approach.

These tools can be classified in two categories. A first category are existing tools for relational databases which are incorporating some NoSQL systems. At this moment, these tools have only added support for document systems, being MongoDB the system integrated in the most of them.

For example, ER/Studio [49] and ERwin [50] provide utilities to extract and visualize schemas for MongoDB and CouchDB since 2015. They extract schemas as a set of entity types whose properties are the union of all fields discovered in objects of that entity, but variations and relationships are not addressed. Recently, ERwin Data Modeler provides an integrated view of conceptual, logical and physical data models to help stakeholders understand data structures and meaning.

The second group is formed by new tools developed with the purpose of offering data modeling for polyglot persistence. As far as we know, *Hackolade* [69] is the only tool that integrates database systems for the four most common NoSQL paradigms as well as a wide number of relational systems and other leading data technologies. Recently, it has been announced the creation of a unified model named *Polyglot Data Model* but no details have been published. Unlike U-Schema, *Hackolade* does not address variation and references in the NoSQL schema extraction. Entities extracted are represented as the union of all the

fields discovered in different variations of the entity. The collision of fields with the same name but different type is not considered but that modeler should make a decision.

DBSchema [43] is a tool similar to Hackolade: It allows the developer to define schemas with a graphical layout, but also to apply a reverse engineering process to an existing database in order to extract the schema, as long as there is a JDBC Java driver for it. Queries can be created in an intuitive way or either using SQL. In this tool, variations are not considered at all, since it applies a SQL approach to infer the schema, in which variations are not taken into account.

3.1.8 Comparison of generic metamodel proposals

Next, we will contrast the metamodel proposed in this thesis with those described above. In this way, we will expose the contributions of our proposal to the state of the art.

GER (DB-Main) Like the GER metamodel [77, 72] for DB-Main, our U-Schema generic metamodel is meant to be the core element of a data engineering tool. However, DB-Main was focused mainly on relational systems, and also on earlier database systems. Instead, we are interested in both structured and semi-structured data, specially in the emerging NoSQL stores and relational databases. U-Schema is intended to represent logical schemas, so that conceptual and physical schemas are separately modeled to have a simpler database schema representation. Because of this concern separation, reusability is promoted, and models are kept simple and readable. Because of this concern separation, reusability is promoted, and models are kept simple and readable. The conceptual and physical metamodels are out of the scope of this thesis, but an exploration of the connection between logical and physical schemas has been performed for MongoDB. Unlike GER, we do not have to define a sub-model of U-Schema for each database system. U-Schema acts as a pivot representation, able to represent NoSQL and relational schemas for all paradigms. The set of rules that maps each data model to U-Schema determines the U-Schema elements involved, and therefore the valid structures. A central notion of U-Schema is structural variation. Variations of entity and relationship types can be represented while GER do not include elements specific of NoSQL stores like structural variations. This information can be useful in different tasks like analyzing the database evolution. We have defined U-Schema with the

Ecore metamodeling language with the purpose of taking advantage of MDE technology integrated in the EMF framework [110]. In DB-Main, a different schema extractor had to be developed for each database system. In our case, a common strategy have been defined which address the scalability and performance issues. In DB-Main, a different schema extractor had to be developed for each database system. In our case, a common strategy have been defined which address the scalability and performance issues.

Universal metamodel . While the universal metamodel of Atzeni et al. is aimed to instantiate data models, U-Schema is a unified metamodel able to represent schemas of a variety of databases. Therefore, the metamodeling architectures are different: Universal metamodel/Data Model/Database Schemas vs. Ecore/U-Schema/Database Schemas. It is worth noting that our approach does not prevent the definition of metamodels for representing any existing data model that is integrated in U-Schema. However, we have considered that creating these metamodels would not provide any benefit as intermediate representation, as the variation schema to data model transformations would be very close to the variation schema to U-Schema transformation. The expressiveness of the Universal metamodel is covered by U-Schema elements. In addition, U-Schema includes the notions of relationship types and structural variations, which are convenient to represent schemas of NoSQL stores, specially graph stores. Atzeni et al. metamodel has not evolved to include elements specific of NoSQL stores such as structural variations. While we have used the EMF metamodeling architecture to create U-Schema, Atzeni et al. had to implement their own metamodeling architecture from scratch, as well reporting and visualization tools. Instead, EMF provides tools supporting model comparison (EMF Compare)[‡] and model diff/merge operations (EMF Diff/Merge),[§] as well as model-transformation languages to implement the *modelgen* operator. Therefore, our availability of tools for basic model operations is larger than the Atzeni et al. solution, an we have not to devote effort to build such tools.

GeRoMe . U-Schema clearly differs of GeRoMe in its purpose and the kind of representation of the generic metamodel. Our unified metamodel has been defined by applying object-oriented conceptual modeling, the technique commonly used currently to create

[‡]EMF Compare website: https://www.eclipse.org/emf/compare/.
[§]EMF Diff/Merge website: https://wiki.eclipse.org/EMF_DiffMerge.

metamodels, and using a well-know metamodeling architecture, while GeRoMe uses *role-based modeling*.

As far as we know, neither of the three generic metamodels here considered (GER, Atzeni et al., and GeRoMe) has evolved to include elements specific of NoSQL stores. Therefore, none of them has addressed the representation of structural variations or relationship types. In the case of GER and Universal metamodels, an extension is not feasible, and new roles could be created in GeRoMe but this would further complicate the metamodel.

NoAM [14] and SOS [15] were also designed with a purpose different to U-Schema. SOS aims to achieve a uniform accessing, and NoAM is part of a design method. Instead, U-Schema has been devised to have a uniform representation able to capture data models of NoSQL and relational data models, with the aim of facilitating the building of database tools supporting several database systems. Therefore, U-Schema offers a higher level of abstraction than SOS and NoAM. These representations are closer to the physical level than the logical. Thus, some key aspects for a logical schema are neglected, such as the relationships between entities. In addition, the existence of structural variations is not considered. Finally, MDE technology was not used in their definition.

ERwin's Unified metamodel . Several significant differences are found between U-Schema and the ERwin metamodel [118]: (i) U-Schema is not only able to represent aggregate-based systems, but also graph stores; (ii) U-Schema is more expressive, ModelSet only includes the three basic constructs of modeling, but this is similar to our variation schemas that are input to the analysis process; (iii) Being U-Schema a representation at higher level of abstraction, the definition and implementation of operations such as schema mapping, visualization, or schema discovery are easier; (iv) U-Schema represents structural variations; (v) Instead of a proprietary tool, U-Schema is part of a free data modeling tool.

TyphonML was designed to be a generic metamodel for NoSQL stores. However, some remarkable differences can be identified between TyphonML and U-Schema: (i) TyphonML was defined as abstract syntax for a schema definition language, instead U-Schema is separated from any language. In fact, we have created the Athena language on U-Schema [35], and other languages could be defined [36] (ii) The existence of structural variation in

NoSQL systems is not considered; (iii) As can be observed in [114], the TyphonML metamodel integrates logical and physical aspects for each paradigm; instead, our choice is to separate both levels of abstraction in two metamodels; (iv) Although graph stores are considered, the concept of relationship type is not included in TyphonML, therefore we are not able to understand como graph schemas are represented; (v) Aggregates are not modeled as a separate concept, but the same metaclass *Reference* represents both aggregates and references by using the boolean attribute *isComposite* to record the kind of relationship; Instead, aggregates and references are two different metaclasses in U-Schema, which allows us to have a complete semantics. The logical elements of TyphonML are limited to *Entities* that aggregate *Attributes* and *References*, while our unified metamodel has a wider and richer set of semantic concepts.

Table 3.3 summarizes the above discussion. Several comparison criteria have been drawn of our analysis of the published proposals: Aim of the generic metamodel, database paradigms that integrates, elements that includes, formalism used to create it, levels of the metamodeling architecture with which is defined, how schemas are instantiated, ability to represent structure variations, levels of schema supported, how schemas extracted are represented, schema extraction strategy applied and whether or not the extraction approach takes into account scalability.

3.2 Schema Extraction

Several approaches to extract schemas from NoSQL document stores have been published [119, 84]. Moreover, some works on schema extraction from JSON datasets have also been presented like [38]. A detailed study of these four works can be found in the thesis of Severino Feliciano [51], where they were contrasted to the document data metamodel designed in that thesis [101]. Also, schema extraction for Neo4j graph databases has been addressed in [39], where data insertion statements are analyzed to discover the implicit schema. While all these works extract schemas from stored data, document schemas are discovered thorough a code analysis in [86]. In that work, the aim is to discover the evolution of the schema. Next, we will present a discussion of the cited works.

	DB-Main	Universal Metamodel	GeRoMe	SOS/NOAM	ERwin Unified Data Modeler (ModelSet)	U-Schema
Aim	Evolution tool	Model Management	Model Management	Uniform access / Database Design	Modeling tool	Database engineering toolkit
Supported Databases Paradigms	Relational, OO, ER and Early databases	Any metadata formalism and database schema	Any metadata formalism and database schema	NoSQL databases	Relational, document, and columnar	Relational and NoSQL (columnar, document, graph, and key-value)
Unified Metamodel	GER based on ER	Set of 13 meta-constructs to cover all data models	Set of 48 roles to cover all data models	Collection, Struct or Block, and Attribute (very near to physical model)	Entity, Property, and Relationship	Set of concepts to cover NoSQL and relational schemas
Metamodeling Language	From scratch	From scratch	From scratch	N/A	From scratch	Ecore (Eclipse Modeling Framework, EMF)
Levels of metamodeling	GER metamodel and restrictions to define data model	SuperMetamodel/Data Model/Schema (Own architecture)	Use of role-based modeling	Abstract metamodel / Instances	Unified metamodel / Models (schemas of data models)	Ecore / U-Schema / Models (schemas of a data model)
Defining concrete schemas	Selection of GER elements and definition of constraints	Model elements are annotated with meta-construct to which belong it	Model elements play one of their roles	Programmatically, Java instances	Instances of the metamodel (proprietary solution)	Instances of the metamodel (use of mapping rules)
Schema Levels	Conceptual, Logical and Physical	Conceptual and Logical	Conceptual and Logical	Very simple uniform representation	Conceptual and Logical (Physical separated)	Logical (Conceptual and Physical separated)
Schemaless supported	Not addressed	Not addressed	Not addressed	Structural variations are supported, but not extracted or represented	Not addressed	Structural variations are modeled
Output	ER diagrams and text	ER diagrams and text	Own visualization	N/A	Unified ModelSet notation	U-Schema models in form of Neo4j graphs
Schema extraction	A schema extractor for each system	Not addressed	Not addressed	N/A	No details provided, except the use of machine leaning and statistics are obtained	Common strategy of 2 steps: MapReduce and U-Schema model building process
Scalability and Performance	Not addressed	Not addressed	Not addressed	N/A	No details provided	MapReduce operation on NoSQL stores

Table 3.3: Comparison of approaches defining a Generic Metamodel.

3.2.1 Meike Klettke, Uta Störl and Stefanie Scherzinger [84]

Meike Klettke et al. defined an approach to extract schemas from document stores in [84], which is based on a strategy aimed to extract DTD schemas from XML documents [92]. Actually, the aim of its research work was to detect and remove outlier data. The algorithm devised to discover the implicit schema analyzes stored objects, i.e. JSON documents, and creates a graph that represents the structure of the documents stored in each collection of the database. Once the graph is completed, the schema is derived from it. Figure 3.9 shows the four steps of the process followed in [84]: (i) Document Selection, (ii) Structure extraction, (iii) Construction of the Structure Identification Graph (SG), and (iv) Generation of the JSON Schema.

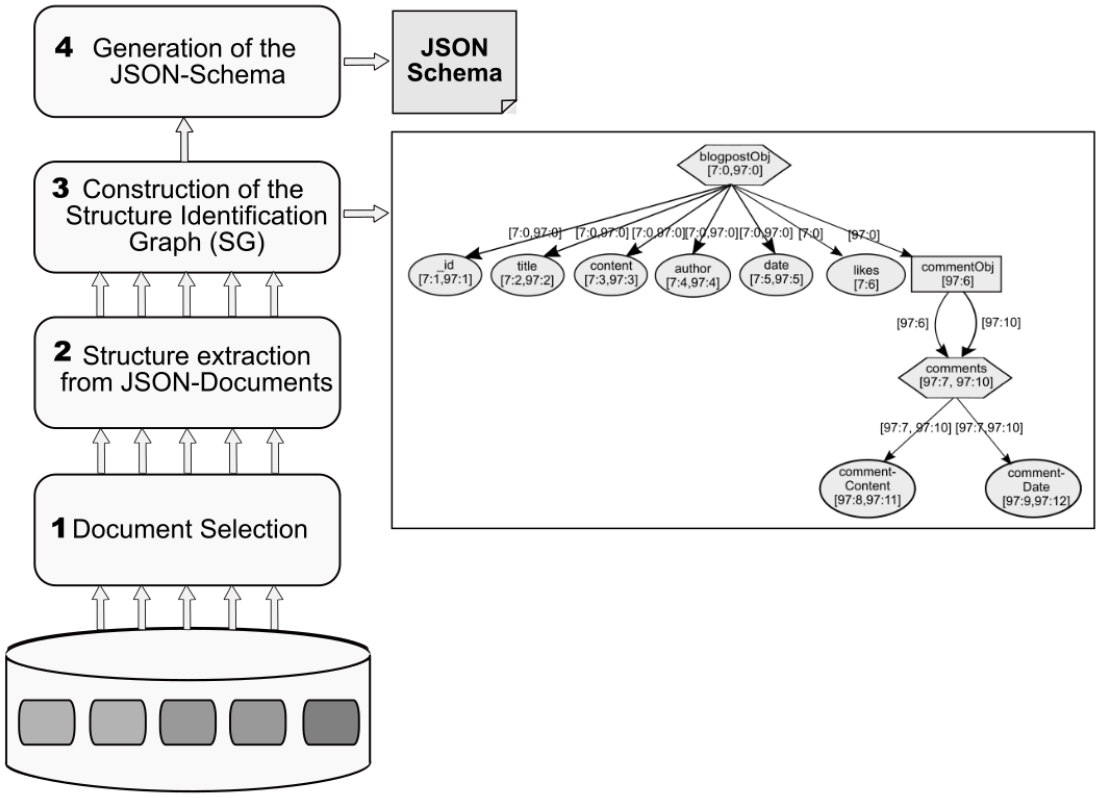

Figure 3.9: Schema extraction process of Klettke et al. (taken from [84]).

In the first step, documents to be analyzed are selected, and users can either select a collection of documents or processing all documents. This selection allows reducing the

execution time if the analysis does not affect all documents. The second and third steps, *Structure extraction from JSON-Documents* and *Construction of the Structure Identification Graph (SG)*, are carried out together, the SG graph is built as the documents are inspected. The extraction algorithm begins by creating the root node and setting an identifier to each document. Then, the algorithm processes each document as follows. First, a top node representing the structure of the document is created and connected to the root node. Then, for each new property or feature found in a document: (i) a node representing the property is created, (ii) the type of the property is recorded, (iii) a graph identifier is created by joining the document identifier along with the node number, and it is recorded in a node list, and (iv) connects the property node to the top node that represents the document structure, these edges also store the identifier of the top node in a list. When a visited property is an object, an additional node is created and connected to the top node. This new node corresponds to the top node of the embedded object. The same processing applied on a root document is recursively applied to the embedded object. If a node already existed for the visited property, the only processing is to create the graph identifier joining the document identifier along with the node number and adding it to the list of identifiers of the node and its edge. In the fourth step, the resulting schema is obtained from the SG graph and represented in JSON Schema format [82]. The SG graph is traversed with a width-first algorithm. The graph identifiers are used to know if a property is required or optional, depending on the property is present or not in all the documents of an entity type. This is calculated by checking whether or not the edge list and node list of identifiers have the same size. When they have the same size, all the documents of a collection (i.e. an entity type) have the property, and therefore it is required. Given a collection of documents, The algorithm groups all the properties found in the same entity type, i.e. a union schema. Two or more properties with the same name can have different types, the algorithm then record all the types found for the property.

Figure 3.10, taken from [84], shows the *Structure Identification Graph* obtained from a collection that contains documents that correspond to posts of a blog. In the figure, the documents are presented as a hexagon, and the properties as circles. The root document (first hexagon) represents a base document that has 7 properties (`_id`, `title`, `content`, `author`, `date`, `likes`, and `commentObj`). The last property `commentObj` is an array of two objects,

these objects are called `comments` and have two properties (Content and Date). All the properties of the example are required except the (`likes` and `commentObj`) properties. Note that the edges incoming to these two nodes have a list of identifiers whose size does not match the size of the list of identifiers of the nodes (hexagons).

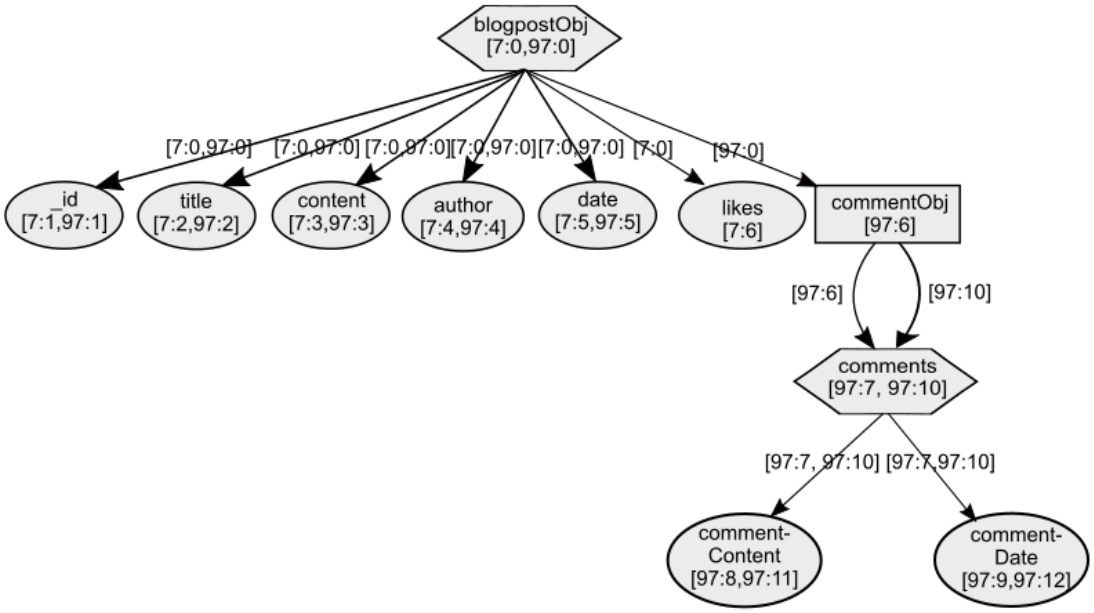

Figure 3.10: Structure identification graph of Klettke et al. (taken from [84]).

The list of identifiers on the edges can be used to obtain some statistics like the percentage of documents that contains a property. These statistics are used to detect outliers as well as missing properties (when very high percentage of documents has a property) or additional properties (properties with very low percentage).

3.2.2 Lanjun Wang et al. [119]

A framework aimed to manage document stores was presented by L. Wang et al. in [119]. These authors observed that well-known datastores have entity types with tens of thousands of variations, which largely complicates the extraction and visualization of schemas. Their work focused on document databases, in particular MongoDB, and a document schema management framework is presented to tackle the problem. This framework includes a set of utilities intended to extract, persist and query schemas along with the presentation of the

schemas. In his research work, they first tested a method based on applying the Canonical Form (CF) algorithm explained in [34]. This algorithm produces schema diagrams in form of a tree, as shown in part (c) of Figure 3.12, which is taken from [119].

The algorithm of Wang et al. processes input documents extracted from a MongoDB database, for example, the sample documents shown in Figure 3.11, also extracted from [119]. For each document, an initial level is created that represents it and for each embedded object a new level is added. A label is associated at each property, which can have two possible structures: "<property name> : <identifier>" for simple properties, or "<property name>,<list of ids> : <identifier>" for embedded objects, where <identifier> denotes a unique identifier assigned to the property, and <list of ids> denote a list of identifiers of properties in the lower level, and identifiers are separated by comma. These identifiers correspond to the properties of the embedded object (lower level).

The part (a) of Figure 3.12 is the result of applying the algorithm to the first document on the left of Figure 3.11. The first level corresponds to the root object that has 3 properties: `article_id`, `author`, `text` whose identifiers, respectively, are 1,2 and 3, and they are part of the level 2. Therefore, the label of the first level is `root,1,2,3:1`. In the level 2, the label of the `author` property is `author,1,2:2` as the value is an embedded document. Whenever a document is processed, if it has a structure different to those previously found, a new label is added on the first level, whose <list of ids> will not match to those of existing labels. That is, each element in the first level corresponds to a structural variation of an entity type. If the new variation has some property that do not exist in the lower level, then it will be added with a new identifier. In this way, the algorithm is applied recursively, and new levels are created if needed.

In Figure 3.12, the first level has three structural variations because S1 and S4 are documents that have the same structure in the example. The S3 document is the most complex, its label includes the identifiers 1,3 and 6 which correspond, respectively, to the properties `article_id`, `text` and `author` of the level 2. Note that level 3 has finally 3 different labels for the property `author`, because this property has embedded objects with 3 different structure.

Due the inefficiency of the Canonical Form (CF)-based algorithm, the authors decided to represent the schemas in another format, in particular the *encoded Schema in Bucket Tree*

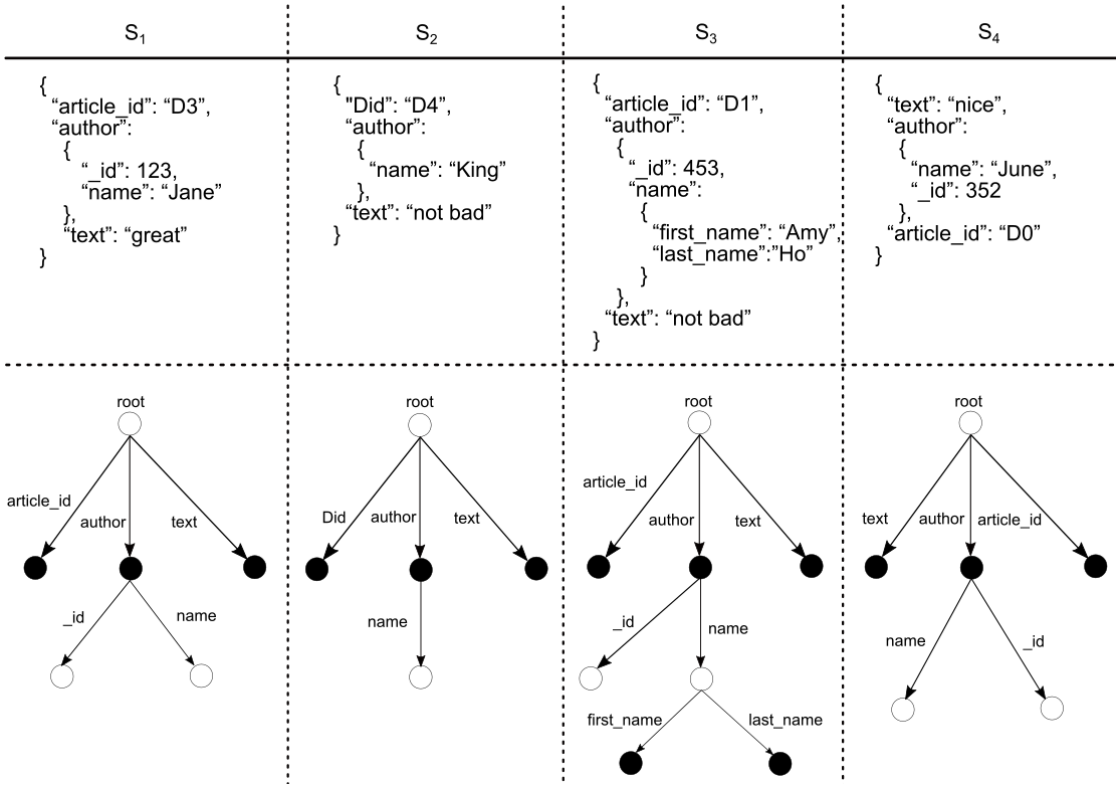

Figure 3.11: Documents Examples of Wang et al. (Extracted from [119]).

(eSiBu-Tree, EST). Figure 3.12 shows the example used to illustrate the EST representation in [119]. The EST-Based Record Schema Grouping algorithm reduces the number of sorts and resizes the map by using a local map for embedded objects instead of a global schematic map.

This new version of the algorithm processes documents in a similar way to the Canonical Form algorithm. For each kind of document, a top node is created that includes a label for each property of the root document; the label is formed by the property name and a unique identifier, which are separated by a semicolon. Embedded document properties are represented by adding a new node that is referenced by the root node, and the property label is formed by the identifier of the aggregate property, as shown in Figure 3.13(a) for the aggregate property `author`: `_id` and`name` properties are in a node referenced from the root node, and their labels start with the number 3 that is the identifier of `author`. A leaf node is created to indicate the end of a nesting branch that corresponds to a particular vari-

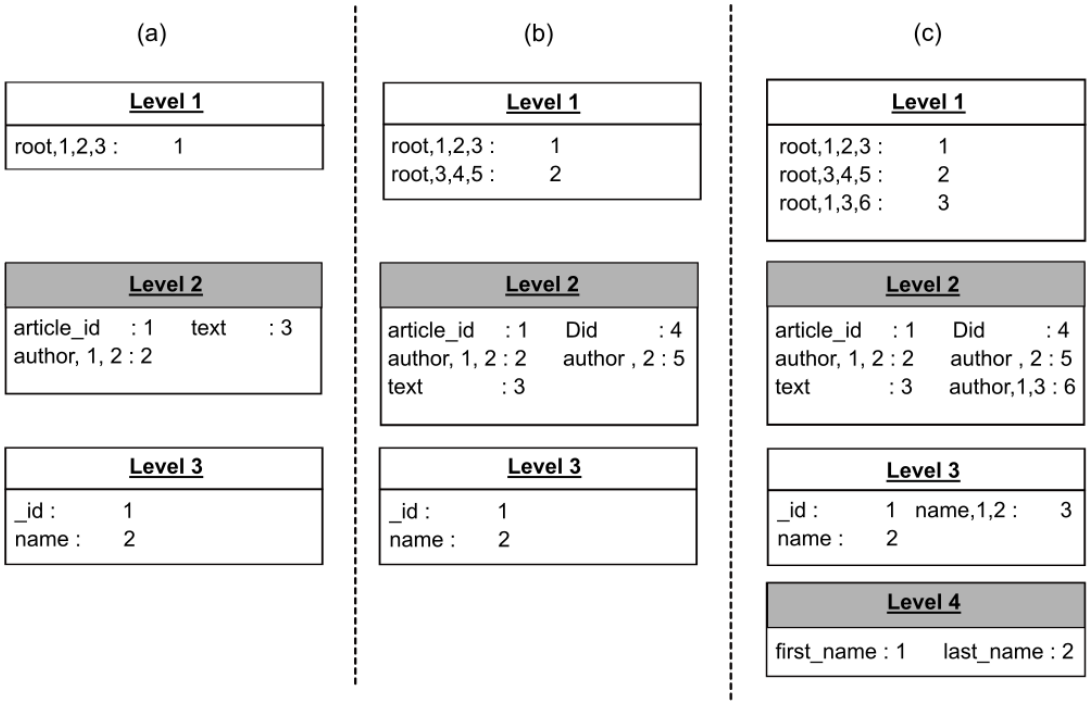

Figure 3.12: Canical Form example of Wang et al. (Extracted from [119]).

ation. Except for the top node (node for the root document), each node has a list with the identifiers of properties in the upper level, and these nodes have a flag that is marked with `T`(rue) in a leaf node, and `F`(false) in the rest of non-root nodes of a branch. Figure 3.13(a) shows the tree obtained if S1 is the first document analyzed; the tree has has only a branch and the nesting level is 1. Figure 3.13(b) shows the tree after S2 is inspected, a new branch is added for the new structure found for the embedded object into `author`, and the nesting level is still 1. Finally, Figure 3.13(c) corresponds to the document S3, the nesting level is 2 because the `name` property in the first branch has an embedded object with two properties: `first_name` and `last_name`.

Efficiency is achieved thanks to the new branches that appear in relation to the previous algorithm, reducing the number of sorts.Searching for documents is optimized by using the list of identifiers in each node. For instance, the root object of S3 has the structure previously found for S1, that is {1,2,3}, then this list is used to find the lower node for the `author` property in order to check if the embedded object structure has changed.

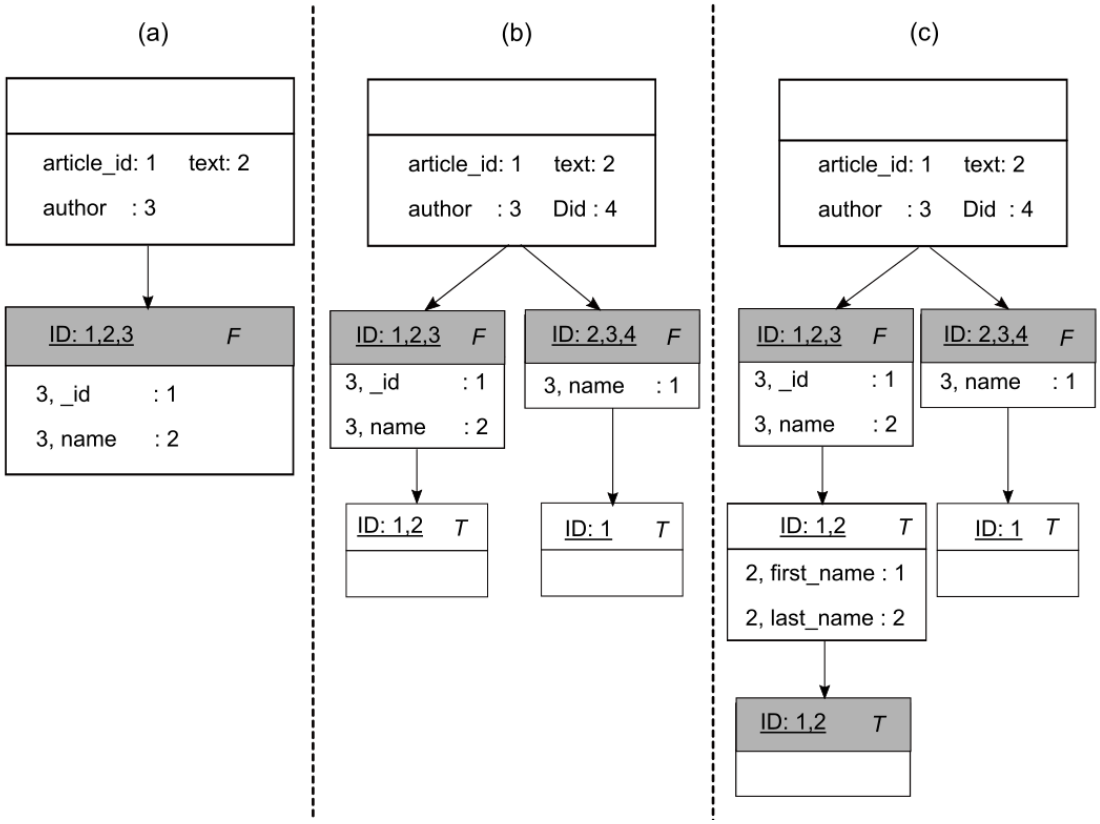

Figure 3.13: eSiBu-Tree example of Wang et al. (Extracted from [119]).

3.2.3 Dario Colazzo, Giorgio Ghelli and Carlo Sartiani [38]

In [38], a two-stages process is presented. In the first stage, a type inference is performed, which uses a MapReduce to obtain a collection of key-value pairs from an input JSON dataset. In each pair, the key is a document specifying the structure or type of a JSON object, and the value records the number of elements of the that type in the dataset. In the second step, similar types are merged by applying a set of heuristics.

3.2.4 Isabelle Comyn-Wattiau and Jacky Akoka [39]

A model-driven reverse engineering approach to extract conceptual graph schemas is described in [39]. The authors apply `CREATE` Cypher statements are analyzed to obtain a graph model: a graph is formed by nodes and edges, and nodes have incoming and outgoing edges. Then, a model-to-model transformation generates an Extended Entity-Relation

(EER) conceptual schema model, whose elements are entities, relationships, attributes, and IS-A relationships.

3.2.5 Comparison of Schema Extraction approaches

The main differences of the schema extraction approach proposed in this thesis with the previously published strategies by Klettke et al. [84], Wang et at. [119], Colazzo et al. [38] are follows. First, they are focused on document stores. Instead, we have defined a general strategy applicable to the four main NoSQL paradigms and the relational model. Second, like the proposal of Colazzo et al. for JSON document, our approach use a MapReduce operation to improve the efficiency of the extraction process; Third, our strategy discovers relationships between entity types, and structural variations of entity and relationship types. Wang et al. obtain structural variations of entities whereas Klettke et al. obtain the union schema of each collection of the database. A union schema has a name and data type pair for each feature that is present in the documents of a collection. These features are classified into two categories: *required* if they are present on all documents contained in the collection, or *optional*. Fourth, the output of our inference process is a model that conforms to an Ecore unified metamodel. In this way, we can take advantage of benefits offered by MDE, which were commented in Section 2.1. Meike et al. use JSON Schema to represent the union schemas extracted whereas Wang et al. use a tree structure to represent all the variations extracted optimizing search operations on the schema. In both works, relationships between entity types are not inferred.

This thesis was conceived from the experience gained in the thesis of Severino Feliciano [51, 101]. Instead of only inferring document schemas, we have addressed the design and implementation of a generic schema management approach: a unified metamodel and a common strategy to extract schemas.

With regard to the work of Comyn-Wattiau and Akoka [39], our proposal differs in several significant aspects, apart from being a generic strategy and use a MapReduce to considerably improve the efficiency: (i) We infer schemas from stored data and database access code , instead of using `CREATE` statements. Comparing with our code analysis approach, we analyze the query and not the creating statements. (ii) Our strategy allows us to obtain the cardinality of each relationship. (iii) We extract a graph logical schema that include

structural variations for relationship and entity types. (iv) The output of the process is a model that conforms to an *Extended Entity-Relation* (EER) metamodel, while we extract a U-Schema model.

3.3 Code Metamodels and Code Analysis

This section is devoted to discuss work related to our code analysis approach devised to extract schemas and apply database refactorings. Firstly, some metamodels proposed to represent GPL code are introduced, and then we will describe some database code analysis strategies aimed to discover the complete schema or some schema elements as foreign keys.

Building generic metamodels to represent application code with the purpose of analyzing it has always aroused interest. Some metamodels proposed are: KDM [68], MoDisco [25] and FAMIX [45]. Regarding research works on database code analysis is remarkable the work performed by the Precise group of the Namur University (Belgium) in the last two decades [86, 55, 85, 88, 103, 87, 94, 19]

3.3.1 KDM Metamodel

The *ADM* (*Architecture Driven Modernization*) [67] initiative was launched in 2003 by OMG to promote the use of MDE in software modernization processes. ADM proposed to create a set of standard metamodels for common tasks in software modernization. *Knowledge Discovery Metamodel* (*KDM*) [68] is the main metamodel of *ADM* since it represents the code of an application at different levels of abstraction. *KDM* is a very large metamodel composed by different packages that allow to define any type of application and language, these packages are used to represent aspects from the code to other levels such as the physical one. Among all the packages, the code and the action packages are used to represent the code and data package to represent the data of the application, even aspects from the code to other levels such as the physical one. In addition, it can be extended with *Micro-KDM* to specify aspects of a specific language.

Figures 3.14, extracted from the specification in [68], shows the main elements of the *Code* package, and Figure 3.15, also extracted from the specification, shows the elements of the *Action* package.

In figure 3.14 you can see the *Code package*. A *CodeModel* represents a software system and is made up of a set of *AbstractCodeElements*. From this class inherit classes: (i) *DataType*, which are extended to represent types, (ii) *ComputationalObject*, extended to represent functions and procedures (*CallableUnit*), or methods (*MethodUnit*), and (iii) *Module*, to represent a software component or artifact.

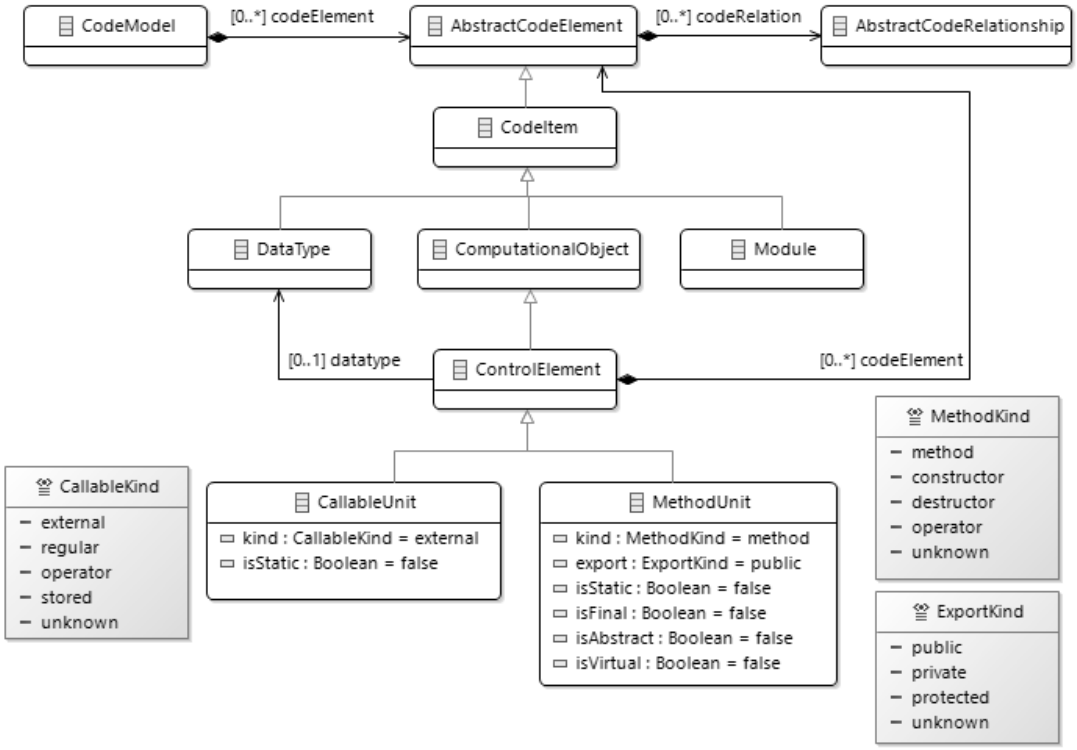

Figure 3.14: Excerpt of the Code Package of the KDM Metamodel (Extracted from the specification in [68]).

In figure 3.15 the main elements of the *Action package* are shown. This package shows the *ActionElement*, which is the main element used to represent the behavior of the code, the *ActionElement* also inherits from the *AbtractCodeElement* of the *Code package*. The elements to direct the flow inherit from *ControlFlow* and are used to represent the flow in conditional statements, and the data is passed between *ActionElements* through *Reads* and *Writes* classes that store the data in a *DataElement*. There is also the *Calls* class that joins an *ActionElement* with a *CallableUnit* representing a function, procedure or method call. Another interesting element is *BlockUnit*, which inherits from *ActionElement*, and is used to represent a block of

code.

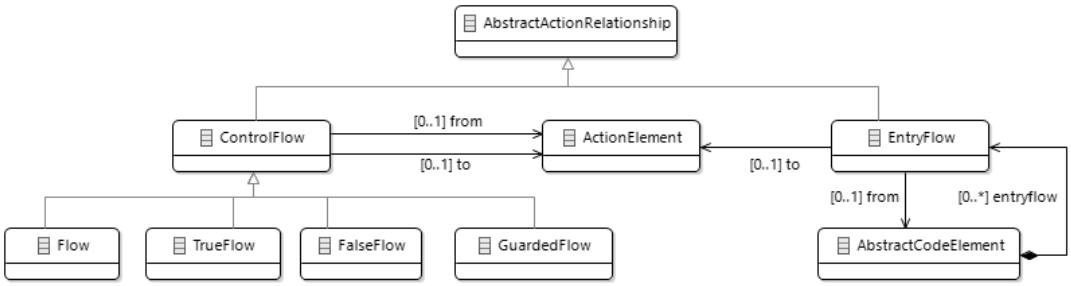

Figure 3.15: Excerpt of the Action Package of the KDM Metamodel (Extracted from the specification in [68]).

In addition to *KDM*, *ADM* includes the *Abstract Syntax Tree Metamodel* (*ASTM*) [63] to represent the code in AST form, the Software Metrics Metamodel (SMM) [66] that represents software metrics and *Automated Function Point* (*AFP*) [62] that defines an algorithm to automate the calculation of function points. All *ADM* metamodels make use of *KDM*.

3.3.2 Modisco Metamodel

MoDisco is a framework defined to modernize legacy applications [24, 25], which is integrated into the Eclipse Modeling Project. Several MoDisco metamodels were defined to represent code of several software languages, for example metamodel for GPL as Java [90] and C# [91]. The framework incorporates several injectors, named *Discoverers*, to parse code and generate models that conform to these metamodels, which can also be transformed into KDM models.

MoDisco metamodels are higher level than *KDM* and easy to read because the metaclasses represent code statements of a particular software language, instead of being generic elements. Figure 3.16 shows an excerpt of the Java Modisco metamodel, most of metaclasses are similar or even the same to the C# metamodel. In these metamodels, an abstract class, named *Statement*, is the root of all classes representing statements of the represented language, e.g., *ForStatement* and *IfStatement*, including *Block* to represent a set of *Statements*. They also have a set of classes to model the expressions, which inherit from the *Expression* abstract metaclass, e.g., *Assignment*, *FieldAccess*, or *ConditionalExpression*. Methods are modeled by means of the *AbstractMethodDeclaration* metaclass and *AbstractMethodInvocation*

represents method calls. Those metaclasses represent common statements of all GPL languages. As we indicate later, our metamodel aimed to represent JavaScript code is inspired by the Modisco metamodels.

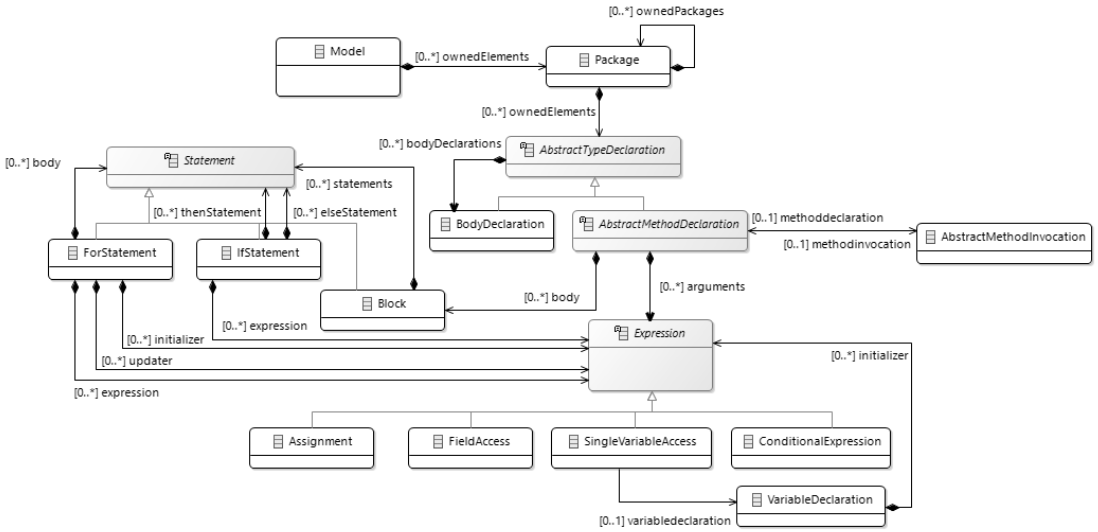

Figure 3.16: Excerpt of the Java Modisco Metamodel (Extracted from [90]).

3.3.3 FAMIX Metamodel

FAMIX is another code metamodel defined to represent code of any object-oriented programming language, which was defined as part of the *FAMOOS* project [46] [45]. Like Modisco framework, FAMOOS aimed to build a tooling to be applied in software reengineering or modernization. FAMIX provided a core metamodel which must be extended to represent a specific programming language. FAMIX has been used in many projects, for instance, it is part of the *Moose*[¶] code analyisis tool [13] or the work described in [112] where FAMIX is used to automate code refactoring.

Figure 3.17 shows the main metaclasses of the core package of*FAMIX metamodel* [46]. Metaclasses for the more common constructs can be observed, as *Namespace* and *Package* to model class groupings, *Class* for class definitions, *Method* for methods, and *Attribute* for properties.

[¶]Moose website: www.moosetechnology.org.

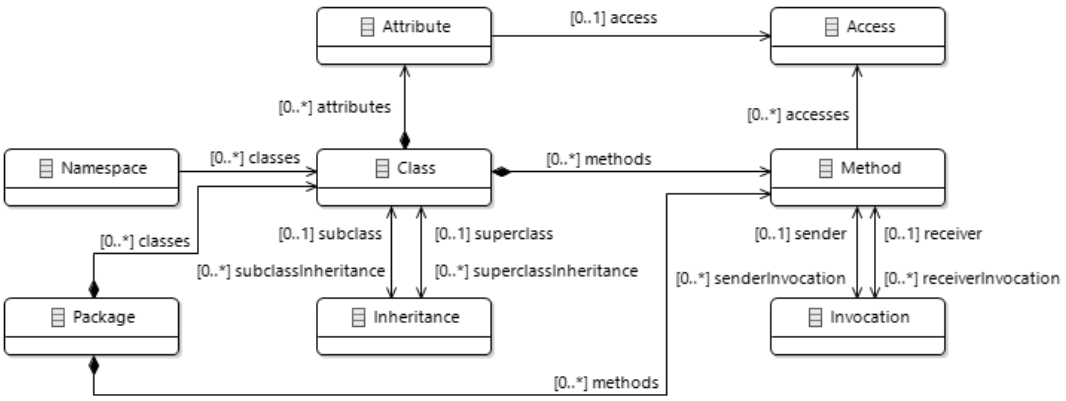

Figure 3.17: Main metaclasses of FAMIX Metamodel (taken from its specification [46]).

Accessing to properties is modeled by the *Access* metaclass, and method calls by *Invocation*. There are several classes to represent the different kinds of variables: *GlobalVariable*, *ImplicitVariable*, *LocalVariable*, *Parameter*, or the before mentioned *Attribute*. The data type concept is modeled by *Type*, which has subclasses as *Class* and *PrimitiveType*. A *FAMIX* extension for Java is provided in the *FAMIX-java* package, which includes metaclasses for handling exceptions as *Exception*, *DeclaredException*, *ThrowException* and *CaughtException*. Metaclasses for representing Enumerates with *Enum* and *EnumValue*, metaclasses *ParameterizableClass* and *ParameterType* to represent parameterized classes and types, and *ParameterizedType* to indicate the type of a parameterizable class or type. CodeCritics is a tool that performs static analysis of PL/SQL code with the purpose of measuring its quality by applying some rules or metrics [44]. This tool uses an extension of *FAMIX* to model SQL code.

3.3.4 Francisco Bermúdez, Jesús García-Molina and Óscar Díaz [103]

Francisco Bermúdez et al. [103] presents a reengineering approach to improve the logical schema of a relational database in the scenario of data migration for data-intensive Java applications. The schema is improved by detecting if foreign keys are missing, or if declared foreign key are not used, and the the schema is also normalized if needed. An MDE approach is applied, and several metamodels are defined: Schema Defect (only Fks missed are considered), Data Definition Language (DDL) and Data Manipulation Language (DML). DDL metamodel is used to represent declared schemas and the DML metamodel to repre-

sent queries and data insert scripts.

The process is organized in the three stages characterizing a re-engineering process: reverse engineering, restructuring and forward engineering. In the first stage, three models are injected from database code, and DDL and DML scripts, which are analyzed to obtain the defect model from three different sources. Such an analysis is implemented by means of a M2M transformation. Regarding the code analysis, the source code is parsed to obtain the String literal containing *Select* queries. Also, if the query is splitted in different strings or variables, the code analysis join them in a single string. When the query string is complete, it is analyzed to discover the tables and condition of the *FROM* and *WHERE* clauses, respectively. The condition is then analyzed to search for join between tables. This information is used to check if all the foreign keys are declared. In the *Restructuring* stage, the schema is modified to remove defects detected ans selected by the administrator. After the schema quality is analyzed, a normalization process is applied if needed Finally, all artifacts necessary to modify the database and code are generated in the *forward engineering* stage, e.g., SQL scripts to update the database and the code of the JPA classes needed to access the new schema.

3.3.5 Works of the Precise Group

Recently, Boris Cherry et al. [33] defined a 3-steps process to find databases accesses in JavaScript applications using Node Mongodb Driver API and Mongoose API. The process first uses CodeQL[*] to parse the JavaScript source code. The code representation obtained is stored in a relational database. This representation uses a language-specific database schema, but there is no a public specification of the schema. The code is accessed using the classes defined in the API.[**] Second, CodeQL queries are issued on that database to search for database access methods. Finally, the methods returned are filtered by applying some heuristics. This filter is necessary because some methods may have same names to the APIs methods.

Jehan Bernard and Thomas Kintziger have addressed the detection of bad smells for NoSQL stores in their Master's thesis [19]. In this thesis, they first define 11 code bad smell

[*] CodeQL website: https://codeql.github.com/.

[**] CodeQL library for JavaScript: https://codeql.github.com/codeql-standard-libraries/javascript/.

of MongoDB database usage, which include simple bad smell as *Storage of empty values* or *Too long attribute names*, and more complex as *Inconsistent order of attributes inside a collection* or Next, a static code analysis is applied to detect bad smell in the code. For this, each code bad smell detector is expressed as a CodeQL rule.

In MongoDB community [40], *Separating data accessed together* pattern refers to accessing two or more collections to obtain data that could be in a single collection. For its detection, Bernard and Kintziger created a rule for searching, in the same function, join queries issued on different collections.

As further work, they commented the extraction of the schema to improve the detection of code bad smells, and they noted that having the schema is necessary for several bad smells. They propose some ways to obtain the schema for different Object Document Mapper (ODM), but they were not implemented.

Jérôme Fink and Maxime Gobert and Anthony Cleve have defined an approach to update queries when schema evolves in hybrid polystores [55], as part of the Typhon project mentioned in the first section of this chapter. This updating requires to analyze the query code. For this analysis, they have considered that polystore schemas are declared with the *TyphonML* language and queries are expressed with the *TyphonQL* language. Both DSLs were defined as part of the Typhon project [85]. A TyphonML script consists of three parts: the logical schema, a mapping from the logical schema to a physical schema for a concrete system, and a list of schema modification operations [114]. The input to the process is therefore (i) the schema modeled with TyphonML, (ii) the queries expressed with TyphonQL, and (iii) the list of schema changes to be applied. TyphonML scripts are parsed to obtain the schema, then the TyphonQL queries are traversed and modified according to the schema changes. The results of the process are the modified queries or unchanged queries if the changes could not be applied to certain queries. The results may also include warnings for some of the modified queries.

Loup Meurice and Anthony Cleve addressed NoSQL schema evolution for MongoDB stores in the work described in [86]. Its approach extracts the physical schema of a MongoDB store by analyzing database code. A process is applied to search for queries, and then query arguments and returned results are analyzed to discover collections, properties of stored objects, and references between objects. This process is applied to different versions

of the application code in a control version system. The evolution of the database schema is recorded in a historical database schema that contains the schema of each database entity (i.e, collection) found: their properties (name and type), as well as possible property renaming and date of insertion or removal of a property. Schema evolution is represented in tabular form, where properties that can cause problems are marked on the table. Different color and icons are used to differentiate the kinds of errors as data corruption, or the types of warnings to developers as renaming of a property or collection.

Loup Meurice, Csaba Nagy and Anthony Cleve applied code static analysis to detect queries in Java applications which access relational databases through three different technologies: JDBC, JPA and Hibernate [88]. They defined a different algorithm for each technology. Because SQL queries can be splitted in different part of the code, the three algorithms applied the *Call Graph* technique to implement an inter-procedural analysis. First, the code is parsed to obtain the abstract syntax tree (AST), then a visitor [58] is applied to traverse the AST and search for queries. In the JDBC algorithm, SQL queries are represented in string format, and they are analyzed to find the table(s) and column(s) used in *FROM* and *SELECT* clauses, respectively. The Hibernate algorithm is an extension of the JDBC algorithm since that Hibernate also allows expressing JDBC-like queries; in this case, the purpose is only to find the location of the queries. The JPA algorithm is similar to the implemented for Hibernate, in fact Hibernate is an implementation of JPA specification. The approach is validated using three real source code applications and could detect from 71.5% to 99% of database accesses.

Loup Meurice et al. defined a strategy to detect referential integrity constraints (RICs) that are implicit in the code and data, but they are not declared [87]. More specifically, their approach discovers FK candidates by means of a 3-steps process where three information sources are analyzed: database schema, stored data, and the program source code. Firstly, PK columns of a table are used to find columns of other tables with similar name. In this way, several FK candidates are detected. In the second step, data are analyzed to measure to what extent each FK candidate is really a FK: number of rows whose columns effectively references to the primary key of the target table. In this way, a a percentage of matching rows is calculated for each FK candidate. Finally, the application code is analyzed depending on the used technology. For JDBC, the SQL queries are represented in strings

and *WHERE* clauses are analyzed to find joins by applying the algorithm commented above for [103]. In the case of Hibernate, the mapping files are used for finding the implicit FKs, and Java annotations are analyzed for JPA, in the same way as Hibernate. The approach was validated wiht a real database of a very large hospital [105].

Csaba Nagy and Anthony Cleve implemented SQLInspect [94]. This tool is aimed to help developers dealing with SQL queries. SQLInspect is able to find and locate SQL queries in Java code and provide some metrics like query complexity, number of queries in a class, or queries with too much joins. Eclipse Abstract Syntax Tree (JDT) is used in the code analysis to search for API calls which issue SQL queries. If the call uses a prepared statement as parameter, then the code is also backward analyzed to find the prepared query. Then, the queries (API calls) found are passed to the *Query Extractor*, defined in [88], to obtain the query in string format. The query strings are parsed and the result is the input to three components: *Smell Detector*, *SQL Metrics* and *Table Access Analyzer*. These components provide metrics and possible warnings on the query.

3.3.6 Comparison of static analysis approaches for code accessing NoSQL stores

The *KDM* [68] metamodel is remarkably complex since that it was designed to represent applications of any domain. In particular, the Code and Action packages are complex because they are GPL-independent generic metamodels. Instead of using these two metamodels, we considered more convenient to create a simpler and more usable code metamodel- Taking into account some existing object-oriented code metamodels, such as FAMIX and Modisco, we have defined a metamodel that integrates the essential constructs of object-oriented languages as Java or JavaScript. As we were analyzing code, the metamodel was extended to achieve the final version that includes aspects as code blocks (lambda expressions in Java 8 or anonymous functions in JavaScript). The simplicity of our metamodel has made it easier to inject code into models. Our metamodel has 55 metaclasses, while the code and action packages sum up 109. In addition, the structure of Code and Action models is most difficult to understand: code statements are represented by using data read and write operations to express how data are moved.

FAMIX [46] and the MoDisco metamodels for Java [90] and C# [91] model the language constructs at the same level of abstraction, and they represent object-oriented code in a

similar way. While Modisco has a metamodel for each language, FAMIX provided a set of core metaclasses along with an extension mechanism to specialize it for a concrete language. In our case, we decided to define a single metamodel by adding metaclasses for language-specific constructs. At this moment, the metamodel has been validated for Java and JavaScript, but more metaclasses could be needed for other languages. FAMIX is used in the CodeCritics tool [44] that only analyzes the SQL code of the applications. Unlike our code analysis stratey, CodeCritics does not extract the schema schema but it is provided as an input.

Next, we will compare our code analysis approach to those exposed in related works. In Loup Meurice et al. [86], the data access code is only analyzed, while we address all the application code. This is because we are interested in extracting schemas and applying refactorings. This, we have devised an MDE solution that represents code by means of two metamodels: Code and Control Flow. We have tackled a non trival database refactoring: removal joins between data containers (e.g. MongoDB collections). Automating this refactoring entails to modify queries and the code that use data returned from the queries.

Boris Cherry et al. [33] presented an code static analysis strategy aimed to find database access statements, which has been applied to MongoDB applications. For this aim, they use CodeQL to query the code. Instead, we represent code in form of models that are programmatically navigated. Injecting code into models, we are able to extract schemas and apply database refactorings, for instance, removing joins to improve the performance of the database. Therefore, the aim of the work of Cherry et al. is more limited than ours. The code analysis of Cherry et al. has been applied in a work of Bernard and Kintziger [19] aimed to detect NoSQL database refactorings. They first identified a list of refactoring that includes the one considered in our work: join query removals that is named "Separating Data That is Accessed Together", a well-known MongoDB schema design anti-pattern. Refactorings are located but their automated application is not tackled.

An MDE-based re-engineering process to automate a database refactoring is defined by Bermúdez Ruiz et al. [103], where static analysis is applied on SQL code. Our approach mainly differs from that work as follows: (ii) We analyze all the code of the application, not only SQL statements; (ii) We extract the schema from application code, instead of analyzing insert statements; and (iii) We update application code, while object–relational

mapping classes are regenerated in the work of Bermúdez et al.

3.4 Schema Query and Visualization

A *data dictionary* is managed in relational database systems to register metadata on the stored data. This metadata includes the database schema and information about physical implementation, security and programs, among other aspects. The SQL-92 standard specifies the structure of data dictionaries in form of tables and views. SQL can therefore be used to recover information from data dictionaries, e.g. queries returning information on the logical schema as tables without primary keys or tables that are not referenced by foreign keys. In fact, data dictionary information is used to visualize relational schemas in data modeling and metadata management tools. In [3], useful queries to explore schema in a set of popular databases can be found.

In the case of NoSQL systems, a schema query native facility is only provided in systems in which a schema may or must be declared, as OrientDB [96] or Cassandra [30]. OrientDB is a multidatabase system (graph, document and object-oriented) that allows to work in schemaless way or declaring schemas. When developers declare the database schema, they can issue SQL queries on the schema, index or storage. These queries return information in form of tables [8]. Cassandra is a popular columnar store which was introduced in Section 2.3.4. It offers a schema query support similar to OrientDB. In both systems, queries are issued on a physical schema.

The ability of performing queries on entity variations is addressed in a work by Wang et al. [119] whose schema extraction has been previously reviewed. Extracted schemas are recorded in a data structure defined as part of the work to facilitate queries on schemas: *eSiBuTree* trees. A SQL-like language is proposed to express queries, but the authors only show a couple of examples: (i) to check if a particular variation, which is specified by a list of properties, exists for a given entity, and (ii) to find which variations of a given entity have a concrete property. A limitation of this proposal is that queries are issued only on entities because relationships between entities are not inferred: references and entities for embedded objects are not inferred.

Other query solutions are available but they does not support relationships or entity variations. Variety [116] is a database tool developed for MongoDB, which provides sup-

port to analyze schemas. The queries are expressed in Javascript code, instead of using a language tailored to query schemas. Apache Drill [56] is a SQL-based query engine that support access to any kind of database (relational, NoSQL, Hadoop, etc.). Drill represents the extracted schema in form of relational schema, so that SQL queries can be issued on it.

As far as we know, the query language suggested by Wang et al. [119] has not been completely defined and implemented yet. SkiQL differs of that approach in several aspects: (i) It is a completely defined language. (ii) It has been devised for NoSQL and relational systems. (ii) Queries are performed on schemas that include relationships either between entities or between entity variations. (iv) Queries can return information on entities, entity and relationship variations, and relationships.

4

The U-Schema Unified Metamodel

In this chapter, we shall present the unified metamodel proposed in this thesis, which we have called U-Schema. The description of U-Schema will be accompanied by the definition of the mappings from the individual data models to U-Schema (*forward mappings*), and in the opposite direction (*reverse mappings*). Since there are no standard specifications for NoSQL data models, before specifying each mapping, a definition of the corresponding data model will be given. Several applications of U-Schema will finally be commented.

4.1 Concepts in the U-Schema unified data model

U-Schema is a unified logical model that integrates the concepts and rules of both the relational model and the four most common NoSQL data models: columnar, document, graph, and key-value. While the relational model is a well-defined data model, there is no specification, standard, or theory that establishes the data model of a particular NoSQL paradigm. In fact, NoSQL systems of the same kind can have significant differences in features and in the structure of the data. We have therefore defined a logical model for each NoSQL category by abstracting from the logical/physical data organization of the most popular stores of each category. This section will present U-Schema, while the logical model defined for each particular NoSQL paradigm will be presented in the section devoted

to describe how that data model has been integrated into U-Schema.

U-Schema includes the basic concepts traditionally used to create logical schemas, which are part of well-known formalisms such as *Entity-Relationship* (ER) [32] and *UML Class Models* [104]: entity type, simple and multivalued attributes, key attribute, and three kinds of relationships between entity types: aggregation, reference, and inheritance. Also, U-Schema incorporates some additional concepts, such as *relationship types* (as they are considered in the graph data model [12]), and *structural variations* of entity and relationship types. Before presenting the U-Schema metamodel, we will define all these concepts. Not all concepts will be present in all of the data models supported by U-Schema. For example, the relationship type is exclusive of the graph model, but conversely, it does not define aggregation.

In data models, an entity type ε is normally characterized by a set $P^{\varepsilon} = \{P_i^{\varepsilon}\}, i = 1 \dots n$ of named properties. Properties can be of several kinds depending on the type of the object or value a property can hold. Three common kinds are: attributes, aggregations, and references. Given a property P_i^{ε}, it would be an attribute if it can take values of scalar type (e.g. Number) or structured type (i.e. Array or Set), and it would be an aggregation or reference if it is associated to an entity type ε' whose objects are, respectively, embedded in or referenced from objects of the entity type ε, to which the P_i^{ε} property belongs. Keys are a special kind of attribute able to record values used as identifiers of instances of entity types.

Graph data models include *relationship types* in addition to entity types. While nodes are instances of entity types, arcs are instances of relationship types, which can have attributes. Hereafter, we will use "schema type" to gather both entity and relationship types.

Schemas play a similar role to types in programming languages. Given a database schema S, only data conforming to S can be stored in the database. Therefore, all data of an entity type E (resp. a relationship type R) will have the same structure, that defined for E (resp. R) in S. In absence of schema declarations, however, data of E and R can have different structure, that is, E and R will have one or more *structural variations*.

A structural variation of a schema type ε is formed by a set of properties $Q^{\varepsilon} \subset P^{\varepsilon}$, and each pair of variations of ε differ, at least, in one property. The set P^{ε} is therefore the union of the sets of properties of each of its variations. The set P^{ε} is commonly called *union schema* of a schema type in a schemaless system. The properties of an schema type can

therefore be classified as *common* or *specific*, depending on whether they are present in all the variations, or in a subset of them. It is worth noting that specifying a schema type as the *union schema*, the information on its structural variability is lost. We have considered convenient to record this structural variability in data models defined for NoSQL databases, and therefore the notion of "variation" will be part of U-Schema.

When structural variations are considered, entity and relationship types can be defined as follows.

- An **entity type** has a name and is formed by a set of structural variations.
- A **relationship type** (only for graph stores) has a name, is formed by a set of structural variations, and refers to both a source and a target entity type.
- A **structural variation** is formed by a set of named properties. The kind of properties depend on the data model, and can be: attributes, keys, aggregates, and references.

Table 4.1 shows the correspondence between concepts of each of the considered database kinds and the logical modeling concepts that we will use in U-Schema. We will use these concepts to abstract a *logical data model* for the most popular systems of each NoSQL paradigm. These models will be presented in sections 2.3.1 to 5.5.

4.2 The U-Schema Metamodel

Data models are commonly expressed formally in form of *metamodels*. A metamodel is a model that describes a set of concepts and relationships between them. That description determines the structure of models that can be instantiated from the metamodel elements, i.e. *a metamodel is a model of a model*. Object-oriented conceptual modeling is usually applied to create metamodels: concepts and their properties are modeled with classes, and reference, aggregation, and inheritance relationships are used to model relationships between concepts. Figure 4.1 shows the metamodel of the U-Schema data model in form of a UML class diagram. Below, we describe this metamodel.

A `U-Schema` model represents a schema formed by a collection of types (`SchemaType`) that can be either entity types (`EntityType`) or relationship types (`RelationshipType`).

Logical modeling concepts	Relational	Columnar	Document	Graph	Key/Value
Schema	Schema	Database or Keyspace	Database	Graph	Database or Namespace
Entity Type	Table	Table with column families	Collection and nested object	Node label	Multirow entities
Relationship Type	Relationship Table	N/A	N/A	Relation type	N/A
Structural Variation	Table (only one variation)	Rows with different structure within column families	Documents with different structure in a collection	Same label with different structure	Multirow entities with different structure
Key	Primary key	Row key	Document key	N/A	Pair key
Reference	Foreign key	Join between tables	Join between documents	N/A	Join between pairs
Aggregation	N/A	Nested object	Nested object	N/A	Nested object
Attribute	Column	Column	Document property	Node and Relation property	Pair Value
Primitive Types	Scalar Types	Scalar Types	Scalar Types	Scalar Types	Scalar Types
Structured Types	N/A	Collections	Collections	Array	Collections

Table 4.1: Mapping between logical modeling concepts and NoSQL/Relational Database Systems.

Both types have two common properties: They include one or more structural variations (`StructuralVariation`), and they can form a type hierarchy (`parent` relationship).

A `StructuralVariation` has an `identifier` and is characterized by a set of *logical* and *structural* features. `StructuralFeature`s determine the structure of database objects, and include `Attribute`s and `Aggregate`s, while logical features specify what identifies an object (`Key`), and which `Reference`s an object has to other objects.

Each attribute has a name and a data type. The data types included are: `Primitive`

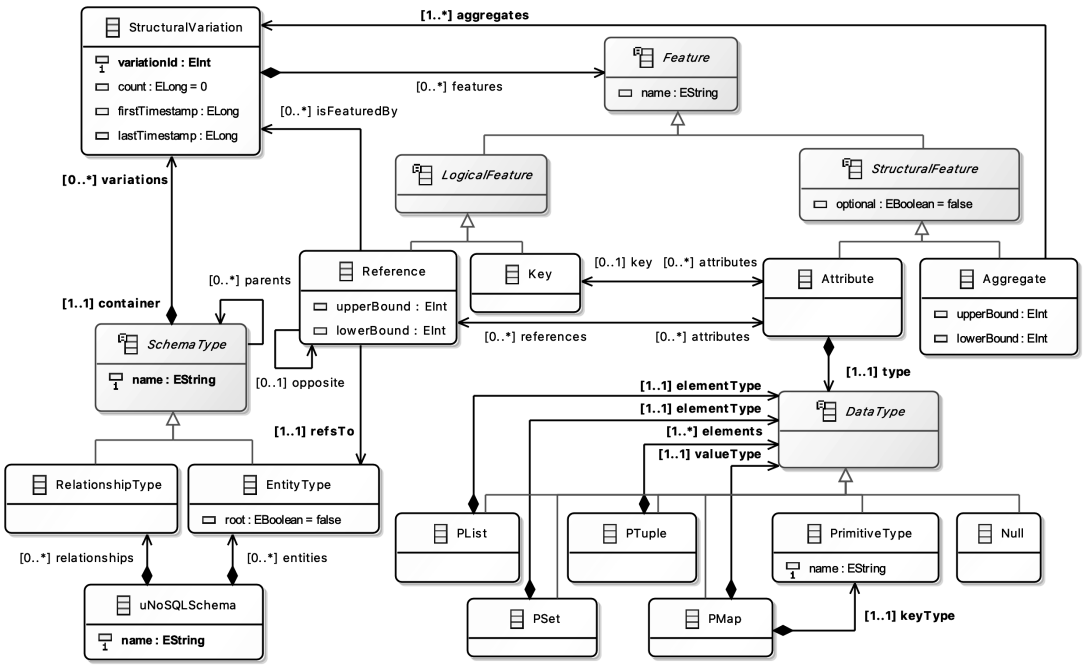

Figure 4.1: U-Schema Metamodel.

(e.g., Number, String, Boolean), *collections* (sets, maps, lists, and tuples), and the special `Null` type. Also, the JSON and BLOB primitive types are included to support relational systems. An aggregation has a name, a cardinality (upper and lower bound), and refers to the structural variation it aggregates, or to a list of variations, if the aggregated object is an heterogeneous collection.

Unlike aggregations, references refer to an entity type (via `refsTo`), and one or more `attributes` that match the set of key attributes of the referenced object (all the variations of an entity type must have the same key.) References also have a name, a cardinality, and an optional inverse reference (`opposite`). Additionally, references can have their own attributes when they represent graph arcs. This entails that a reference has to specify which variation (of its `RelationshipType`) its set of attributes corresponds to (`isFeaturedBy`). `Key` represents the set of attributes playing the role of key for an entity type, holding a unique set of values for each element of the type. `Reference` also points to the set of attributes that form the referenced key (`attributes` property).

The aggregation relationship allows objects to be recursively embedded, then forming

aggregation hierarchies. In these hierarchies, the identification of the root element is important. Thus, an entity type includes a boolean attribute named `root` to indicate whether or not their entities are aggregated by others (`aggregates` relationship).

U-Schema also records information that could be useful to implement some database tasks. For example, as shown in Figure 4.1, `StructuralVariation`s have a `count` attribute to record the number of objects that belong to each variation, and two timestamps that hold the creation dates for the first and last stored object of a variation (`firstTimestamp` and `lastTimestamp`).

The U-Schema metamodel has been defined with the Ecore metamodeling language. Ecore is the central element of *Eclipse Modeling Framework* (EMF) [110], a framework widely used to develop Model-Driven Software Engineering (MDE) solutions [23]. EMF-provided tools such as model transformation languages, model comparison and diff/merge tools, or workbenches for the creation of domain-specific languages (DSLs) could be used to build database tools based on U-Schema models. Metamodeling has traditionally been applied to define data models, and transformational approaches have been proposed to tackle problems involving schema mappings [71, 20]. However, the database engineering community has paid little attention to MDE techniques and tools, although metamodeling and model transformations foundations are well established in the MDE field. Using Ecore, we obtain two benefits: leveraging the EMF tooling to develop database utilities, and favor their interoperability with other tools [103].

4.3 U-Schema Flavors: Full Variability vs. Union Schema

U-Schema allows to accommodate the definition of two model flavors:

- **Full Variability**: All structural variations of all entity and relationship types are stored.
- **Union Schema**: Only one structural variation is stored for each schema type. Structural variability is recorded by using the `optional` boolean attribute of `Feature` to indicate if a feature is present or not in all the objects of an schema type. Union schemas are the schemas normally obtained in NoSQL schema discovering processes [84, 119], and visualized in NoSQL modeling tools.

Note that it is easy to convert a U-Schema model from the *Full* to the *Union* flavors. This conversion loses information, and thus it is not reversible: Given a schema type t with a set of n variations $t.variations = \{V_i^t\}, i = 1 \dots n,$, then t will be replaced by a schema type s with the same name ($t.name = s.name$), and the set $s.variations$ will have a single variation W^s with $W^s.features = \biguplus_{i=1}^{n} V_i^t$, where $\biguplus$ is a function that returns the union set of all the features of all the variations with the following rules:

1. If the same structural feature appears in all variations V_i^t, then it is included in the result set with its `optional` attribute set to false (common structural features).

2. Each structural feature that appears at least in a variation is included in the result set, but with its `optional` attribute set to true.

3. Structural features that appear with the same name (`name` attribute of `StructuralFeature`) but with different type (they belong to a different sub-metatype of `StructuralFeature` or have different values of their attributes), are included with a numeric identifier appended to their `name`, and with their `optional` attribute set to true.

Example of an union schema for the running example presented in Section 2.2 is shown in Figure 5.4. `StructuralVariation`s are omitted for clarity, and optional features are shown in *cursive* and green color.

We will use the Full Variability flavor through the rest of the article, as it contains more information and can be trivially converted to the Union Schema if desired.

4.4 Mappings between U-Schema and the Logical Data Models

A unified metamodel is intended to represent all the concepts of the individual data models that it integrates. Therefore, a mapping must be established between the unified metamodel and each data model. We will call *forward mapping* to a mapping from a NoSQL or relational model to U-Schema, and *reverse mapping* to a mapping in the opposite direction.

As indicated above, we had to define a logical data model for each NoSQL paradigm. As most NoSQL databases are schemaless, the schemas are implicit in data and code. Therefore, the implementation of a forward mapping must first capture all the logical information of

the implicit schema, as described in Section 5.1, and then apply the mapping to obtain the U-Schema schema (i.e., a U-Schema model).

As U-Schema is richer in concepts than each individual data model, forward and reverse mappings are not unique for a particular data model. In addition, U-Schema concepts not present in a specific data model could be mapped in different ways in a reverse mapping. This has led us to introduce the notion of *canonical mapping*. A canonical mapping satisfies two conditions:

1. It must be *forward-complete*, that is, the rules must correctly map all the characteristics of the data model to U-Schema concepts.

2. As a consequence, it must be trivially *bidirectional within a data model*. This is because given a U-Schema model, the original database schema could always be reproduced (as the U-Schema model holds all its information.)

While the canonical mapping rules cover the characteristics of each of the logical data models, there may be cases where a reverse mapping has to be performed on a U-Schema model that contains elements not present in a given data model. Specialized forward and reverse mappings could also be defined for each data model, and even for a given database implementation. These mappings could be devised for specific needs within a development such as a database migration that involves different source and target data models. The need for these mappings raises the interest in creating a mapping language able to specify how the constructs of a given database paradigm are translated into the abstractions of U-Schema, and vice versa. This is out of the scope of this work.

In the following Chapter, the common strategies devised to implement and validate all the forward mappings will be described. In sections 2.3.1 to 2.3.4, we will define a logical model for each database paradigm addressed and formally express the *canonical mapping* between each data model and U-Schema. Additionally, reverse mapping examples will be shown for characteristics not supported in each of the data models. For each paradigm, the forward mapping implementation and validation will be commented. Here, we will introduce the notation used to define the mappings.

- A mapping between an element *u* of U-Schema and an element *m* of a data model is

expressed as:

$$u \leftrightarrow m \| \{\text{list of } \textit{property relations}\}$$

where *property relations* are expressed as indicated below, and the $\leftrightarrow$ operator is commutative.

- A property relation $p_1 = p_2$ expresses that a property p_1 of u and an property p_2 of m have the same value. The $=$ operator is commutative.
- A property relation $p \leftarrow v$ expresses that the value v is assigned to the p property of u or m.
- Let e_1 be a property of u and let e_2 be a property of m, a property relation $e_1 \leftrightarrow e_2$ expresses an enclosed mapping between e_1 and e_2.
- In a property relation that expresses a mapping between two elements, the $map(e, t)$ function can be used to obtain the target element of type t that maps to the source element e; if e maps to a single target element, then the second argument is optional.
- Given an instance of a meta-class in U-Schema, dot notation is used to refer to its properties. For example, given an instance e of `EntityType`, $e.name$ refers to the attribute `name`.
- Functions will be applied on elements of the data model to obtain the value of their properties. Functions will have the same name as the property. For example, given an entity type e, $name(e)$ will refer to its *name* property. Other functions will be introduced in some rules, and their proper definition will be shown.

4.5 Applications of the U-Schema Metamodel

The usefulness of generic metamodels is well known, and has been extensively discussed in the database literature for more than 30 years. In this section, we shall show how U-Schema metamodel can be used to define generic solutions that involve SQL and NoSQL systems. We will first outline an approach to build a generic query language. Next, we will describe a data migration process and analyze how U-Schema facilitates such migrations. Finally, it

will be briefly commented the usefulness of U-Schema to query schemas, generate synthetic data for testing purposes, and visualize schemas in a data format-independent way.

4.5.1 A U-Schema-based Generic Query Language

Given the widespread usage of different data models, developers and companies face the problem of managing several query languages. Therefore, there exists a great interest in creating a universal query language for the variety of data managed in modern applications, and some proposals have recently appeared. PartiQL [2, 7] is a query language created in Amazon to achieve independence of format and data store in accessing the variety of data stored in the company. The language is built on a generic data model able of representing tabular, nested, and semi-structured data. Also, a model-independent query language is convenient in multi-model systems, as it is the case of OrientDB [6]. Both languages are based on the SQL standard due to its widespread adoption.

Main design issues

The U-Schema data model could be used to create a generic language to manage NoSQL and relational stores. This language would have specialized components to manipulate data and to create and evolve schemas. Here, we will focus on the data query. Next, the main design issues that arise in the building of such a language are dealt with.

- *Data representation*: Data returned as result of a query and data inserted into a database must be represented in some format. In a way similar to PartiQL, a JSON-based format could be used to represent data. JSON should be extended to represent specificities of U-Schema as collection types and references.

- *SQL extension*: Because SQL is a very popular language, most query languages for post-relational databases (e.g., object-oriented, NoSQL, spatial) have been defined as extensions of the SQL standard. In fact, generic languages such as PartiQL and OrientDB have also been created as extensions of SQL-92. In this section, query examples will be shown in a SQL-like syntax, and the language defined for U-Schema could also be a SQL extension similar to that defined in OrientDB, which includes document, key-value, and graph data models.

- *Navigation through objects*: Like most systems supporting embedded objects, an aggregation hierarchy specified by U-Schema could be navigated by using the dot notation. To navigate through references, a different notation should be defined to allow a more natural graph-based navigation. Also, a way to access attributes of references should be provided.

- *Graph queries*: An ISO project* was recently launched with the aim of integrating a graph query language (GQL) in the SQL standard. Also, OrientDB extended SQL-92 with functionality to query graphs. These extensions could be appropriate to design the manipulation of graphs on U-Schema. Since U-Schema includes entity types and relationship types, queries could be issued on both types (i.e., all the nodes that are instances of a type).

- *Issuing queries on variations*: Since structural variations are part of U-Schema models, it should be possible to issue queries on one or more variations of an entity or relationship type, instead of being issued on the union type as it occurs by default. Variations could be either extensionally identified or assigned identifiers when the schema is described.

- *Describing, creating, and evolving schemas*: A query language is accompanied of a schema declaration language and a schema operation language. Schemas could be explicitly specified or either inferred from the database.

The U-Schema query engine would be implemented with a strategy similar to that applied for PartiQL [2, 7], as illustrated in Figure 4.2.

Such an engine could work as follows. Queries could be issued from an interactive tool or either been specified as part of programming code (e.g., Java or Kotlin) via a library. A parser would create the abstract syntax tree of the query, and a compiler would traverse the AST to express the query in an abstract intermediate format. In addition to the query, the schema must be an input to the compiler. This schema could be explicitly defined or either inferred by applying the extractors presented in previous sections. Once queries are compiled, an evaluator would issue native queries on a concrete data store, which would

*GQL website: GQLStandardwebiste:https://www.gqlstandards.org/.

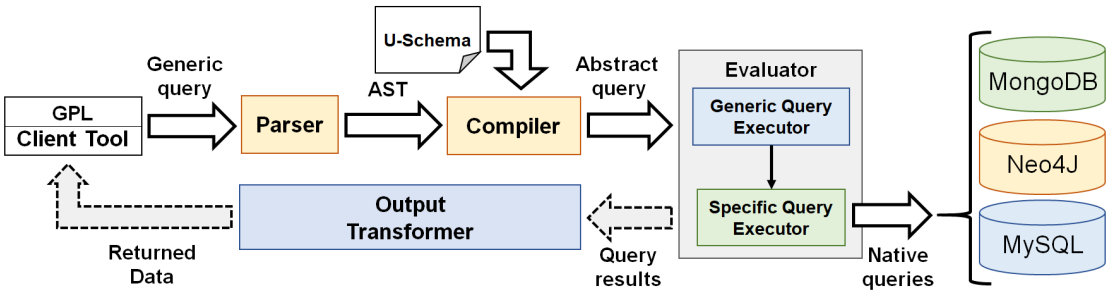

Figure 4.2: Overview of a generic query architecture based on U-Schema.

work in two steps: first, the abstract query would be converted into a specific query for a data model, and then the native query for a particular database is generated and issued. Finally, a component is in charge of receiving the results returned by the database system and transforming them in the expected output representation, i.e. a JSON-like format, as indicated above.

The query language

Now, we will show some query examples to illustrate how the design issues considered above could be addressed in a U-Schema-based query language. The query examples will be written for the schemas inferred from the databases instantiated for the running example. A SQL-like syntax will be used to express the queries. We will focus on how the particular abstractions of our unified data model could be part of the queries.

Embedded objects and References: Navigation and Serialization In the running example, *Address* and *WatchedMovies* objects are embedded into *User* objects in the case of aggregate-based data models. The query below could be written to return the email and the list of watched movies of those users whose address has "`Aylesbury`" as their city. The variable `u` is used to more clearly show the navigation using the dot notation.

```
SELECT u.email, u.watchedMovies
FROM User u
WHERE u.address.city = "Aylesbury"
```

Q1. Email and watched movies of users who live in "`Aylesbury`".

Regarding to the serialization of the result, it could be returned an array of JSON-like documents with the two fields selected of *User*. The value of *watchedMovies* would be an

array with embedded objects that have two fields according to the schema inferred for the running example in Section 5.3: the number of starts and a reference to the watched movie. The reference values could have a special format so that references can be correctly manipulated in the code that receives the result. This format could be "`$ref<entity type referenced>(value)`", and the query result would be serialized as:

```
{
  email: "alison@gmail.com",
  watchedMovies: [
     {stars:3, movie_id: $ref<Movie>(202)},
     {stars:5, movie_id: $ref<Movie>(295)}
  ]
}
```

Note that *User* could denote a document collection, a keyspace, or a columnar table. But this query should be statically incorrect for the graph and relational schemas defined for the running example. This is also true for the queries Q2 and Q3 presented below.

The following query shows how collections could be filtered by applying conditions on their elements. The query returns the name and email of those users that marked a movie with 5 stars once at least.

```
SELECT DISTINCT email, name
FROM User
WHERE EXISTS(watchedMovies[stars = 5])
```

Q2. Name and email of users who marked a movie with 5 stars.

Navigation through references could be expressed as illustrated in the query Q3, which returns name and email of those users that watched the movie titled "`The Matrix`". The "`*`" dereferencing operator is used to access the object that a reference points to. In the query, `(*movie_id).title` denotes the `title` field of the `Movie` object referenced from the `movie_id` field of a `WatchedMovie` object aggregated to an `User` object.

```
SELECT DISTINCT email, name
FROM User
WHERE EXISTS(watchedMovies[(*movie_id).title = "The Matrix"])
```

Q3. Name and email of users who watched a titled "`The Matrix`".

Queries on graphs: Navigation and serialization To query U-Schema models that come from graph stores, it should be considered that they include entity and relationship types,

and references can have attributes as they are instances of relationship types. Therefore, the language should distinguish three kinds of accesses: (i) Given a node, the access to its outgoing and incoming relationships; (ii) Given a reference, the access to its attributes; (iii) Given a reference, the access to the referenced object. The following query examples will illustrate each kind of access. Recall that *User* nodes are connected to *Movie* nodes through *WATCHED_MOVIES* and *FAVORITE_MOVIES* relationships, and also *User* nodes are connected to *Address* nodes through *ADDRESS* relationships, as indicated in Section 2.3.1.

The query Q4 would obtain those users that watched the movie titled "`The Matrix`" (*title* is an attribute of the *Movie* entity type) and marked some movie with zero stars (*stars* is an attribute of the *WATCHED_MOVIES* relationship type). The "`->`" symbol is used to navigate to the destination of a reference (a *Movie* in this case), and we use the dot operator to access the reference itself, and its *stars* attribute. Other operators could be defined to navigate the graph, such as the "`out()`" operator defined in OrientDB.

```
SELECT *
FROM User u
WHERE EXISTS (u->WATCHED_MOVIES[title = "The Matrix"]) AND
      EXISTS (u.WATCHED_MOVIES[stars = 0])
```

Q4. Users who have watched the movie "`The Matrix`" and marked a movie with zero stars.

With regards to the serialization of references in graphs, they may contain attributes, and belong to a specific variation of a relationship type. This information is added to the "`$ref`" type shown above for non-graph references. Additionally, they can include special keys for specifying the source and target elements of the reference. The query below is similar to Q1, but returns *WATCHED_MOVIES* references instead of *WatchedMovies* aggregated objects.

```
SELECT u.name, u.WATCHED_MOVIES
FROM User u
WHERE u->ADDRESS.city = "Aylesbury"
```

Q5. Name and watched movies of users who live in "`Aylesbury`".

Note that we use the dot notation to return the reference itself instead of the referenced *Movie* object. According the format commented above, the query could return a set of JSON-like documents like the following:

```
{
  name: "Allison",
  WATCHED_MOVIES: [
     $ref<Movie,WATCHED_MOVIES~1>({stars:3, $target: 202}),
```

```
    $ref<Movie,WATCHED_MOVIES~1>({stars:5, $target: 295})
  ]
}
```

In this case, "`WATCHED_MOVIES~n`" refers to the given variation that describes the set of attributes of each variation. In this case, the one that has the *stars* attribute. The special "`$target`" attribute holds the actual reference.

Queries on graph schemas could be issued on relationships and/or return relationships, as the query below illustrates. This query traverses all the relationships of type *WATCHED_MOVIES* and returns the name of those users who marked a movie with zero stars once at least. The `target()` operator (equivalent to "`*`") and the `source()` operator would allow the target and source nodes of a given reference to be obtained.

```
SELECT DISTINCT source(w).name
FROM WATCHED_MOVIES w
WHERE w.stars = 0
```

Q6. User names who marked a movie with 0 stars.

Queries and variations When the schema is defined, each of the variations can have their own identifiers. We used `WATCHED_MOVIES~1` above. It would be convenient to define a notation intended to specify intensionally variations instead of using numeric identifiers. For example, the following query would only be applied on *User* variations not having the *favoriteMovies* attribute.

```
SELECT *
FROM User - {favoriteMovies} u
WHERE u.address.city = "Aylesbury"
```

Q7. Users who have no favorite movies and live in "`Aylesbury`".

The query uses an structure-based expression similar to that defined by the research group in the Deimos language [75] to specify the elements of *User* that do not have the given attribute. Finally, other operations related to the management of variations themselves, for example, homogenizing all the variations of a given entity type into just one, are not shown, but are described in the operations defined in the Orion schema evolution language by the ModelUM research group in [36].

4.5.2 Database Migrations

Database migration is a typical task in which a unified or generic representation provides a great advantage. Given a set of m database systems, the total number of migrators required is $m + m$ instead of $m \times (m - 1)$. Here we will describe how U-Schema models can be used to help migrate databases when the source and target systems are different.

To perform a migration, the source and destination databases have to be specified, as well the mapping rules that determine how source data are moved to the destination database. A migration tool usually has to read all the data in the original database, perform some processing, and write the resulting data in the destination database. These steps can be carried out in different ways, that can be simplified by using U-Schema models, as they contain all the information of entities, attributes, and relationships. Therefore, the U-Schema model has to be obtained prior to the aforementioned steps.

There are several options when reading the original data. A set of queries could be constructed to extract the data guided by its structure (i.e., its schema). The inferred U-Schema model from the source database can be used to automatically generate those queries. The queries can produce a set of interchange format files (e.g. JSON or CSV) or can act as a source feed for a streaming process. Likewise, U-Schema models could automate the data ingestion procedure using bulk insertion utilities from files, generated insert queries, or even help to build the ingestion as the last stage of a streaming process.

The next step is to specify and execute the mapping rules between source and destination elements. The mapping rules introduced in sections 5.2 to 5.6 should be adapted to the specificities of the migration. For example, an alternate mapping could be devised for characteristics not present in the destination data model, as was the case we showed with aggregates in a graph data model in Section 5.2.1. The migration rules could be hardcoded, or either specified with a ad-hoc language. This language would be defined taking into account the abstractions of U-Schema. The migration rules would include the U-Schema source element, the target data model element, and the mappings between the parts that constitute the source and target elements, similarly to how we expressed the canonical mappings before.

4.5.3 A universal schema definition language by using U-Schema

Sometimes it is interesting to create a database schema to be able to perform efficiency tests before performing a data migration or even starting the development of an application where execution time is important. For this reason, it could be interesting to create a language capable of creating U-Schema models that, together with other data generation tools such as the one proposed above, can be used to perform performance tests on a synthetic database. A first version of the U-Schema-based schema definition language is defined in [35].

4.5.4 Generation of datasets for testing purposes

Automatic database generation is a point of interest in designing, validating, and testing of research database tools and deployments of data intensive applications. Often, researchers in the data-engineering field lack of real-world databases with the required characteristics, or they cannot access them.

Some works have addressed the generation of synthetic data on relational systems, and some restriction languages have been proposed to this purpose [26, 108]. With U-Schema, a database paradigm-independent restriction language could be defined to tailor the generation of data. In this way, a given specification could be used to generate data for different databases. Note that the language constructs would be at the level of abstraction of U-Schema, and not aligned to elements of any concrete paradigm.

This is of special importance in the case of distributed systems, as most NoSQL deployments are. In this context, a cost model to evaluate query efficiency is very difficult to build, given all the variables involved [89]. Generating different sets of data with different characteristics can help fine-tuning application intended queries. For example, just changing the relationships between the entities of a schema (for example, changing references into aggregations or vice versa), new data that follows this change could be generated to test the queries, helping the developers to find opportunities for optimization.

Finally, another advantage of our approach around U-Schema is that in the case of existing databases, their schema can be inferred into a model, and then used to generate data that can be for the same or different databases, matching the schema or even introducing changes, either for performance tuning or for testing purposes. An initial version

of a U-Schema-based data generation language with the described characteristics is addressed [75].

4.5.5 Definition of a Generic Schema Query Language

Schema query languages help developers to inspect and understand large and complex schemas. In the case of relational systems, SQL is used to query schemas represented in form of tables in the data dictionary. In NoSQL stores, a similar query facility is provided by some systems that require to declare schemas, for example Cassandra [30] and OrientDB [96]. In the case of schemaless NoSQL systems, the number of variations can be very large in some domains, for example 21,302 variations for the *Company* entity type of DBPedia are reported by Wang et al. [119]. Using U-Schema, a generic query language could be defined which would allow querying relationships and structural variations for any kind of NoSQL store, unlike existing solutions. As far we know, querying variations has been only addressed in the mentioned work of Wang et al. [119], which focused on MongoDB, and only suggested a couple of queries to illustrate the idea.

The U-Schema query language allows to query the schema of any type of database system under a unique language, and even make it possible in scenarios where the data is stored in different database systems (polyglot persistence). Some examples of the most common queries that a developer might need are: (i) get an overview of the entities and the relationships between them, (ii) search variations with a set of properties, (iii) check all shared properties of all variations of a specific entity. The results of the queries could be displayed as text or a graphic representation in the form of tables, graphs or trees (hierarchical data). This is addressed in Chapter 7.

4.5.6 U-Schema Schema Visualization

When schemas are extracted they must be expressed in a graphical, textual, or tabular format to be shown to stakeholders. Normally, they are shown as a diagram (e.g., ER or UML). It is possible to take advantage of U-Schema to define common diagrams for logical schemas taking into account the existence of variations if needed. Moreover, U-Schema could be mapped to other formats with the purpose of visualizing schemas in existing tools. This is also addressed in Chapter 7.

5

Strategies for the Extraction of U-Schema Models

Once the U-Schema metamodel and its mapping were presented in the previous chapter, we will describe here the common strategies devised to implement and validate the mappings. In this chapter, we will first explain how U-Schema models are obtained from NoSQL databases or relational schemas. Then, a conceptual schema will be presented as a running example to be used to illustrate the explanations of the following five sections. Finally, the experiments used to validate the U-Schema model building process will be exposed. The implementation and validation strategies are common for all the paradigms, but some stages or experiments are not required in the case of the relational model.

5.1 A Common Strategy for the Extraction of U-Schema Models

As indicated in Section 4.4, in the case of NoSQL stores, applying a forward mapping first requires inferring the schema that is implicit in the data and code. These schemas conform to the *logical data model* abstracted for each NoSQL paradigm. Therefore, U-Schema models are built in a 2-stage process, as illustrated in Figure 5.1. First, a MapReduce operation is performed on the database to infer its logical schema. This stage is not needed for the

relational model. In the second stage, the forward mapping rules are applied to create a U-Schema model from the previously inferred schema. Next, we explain these two stages.

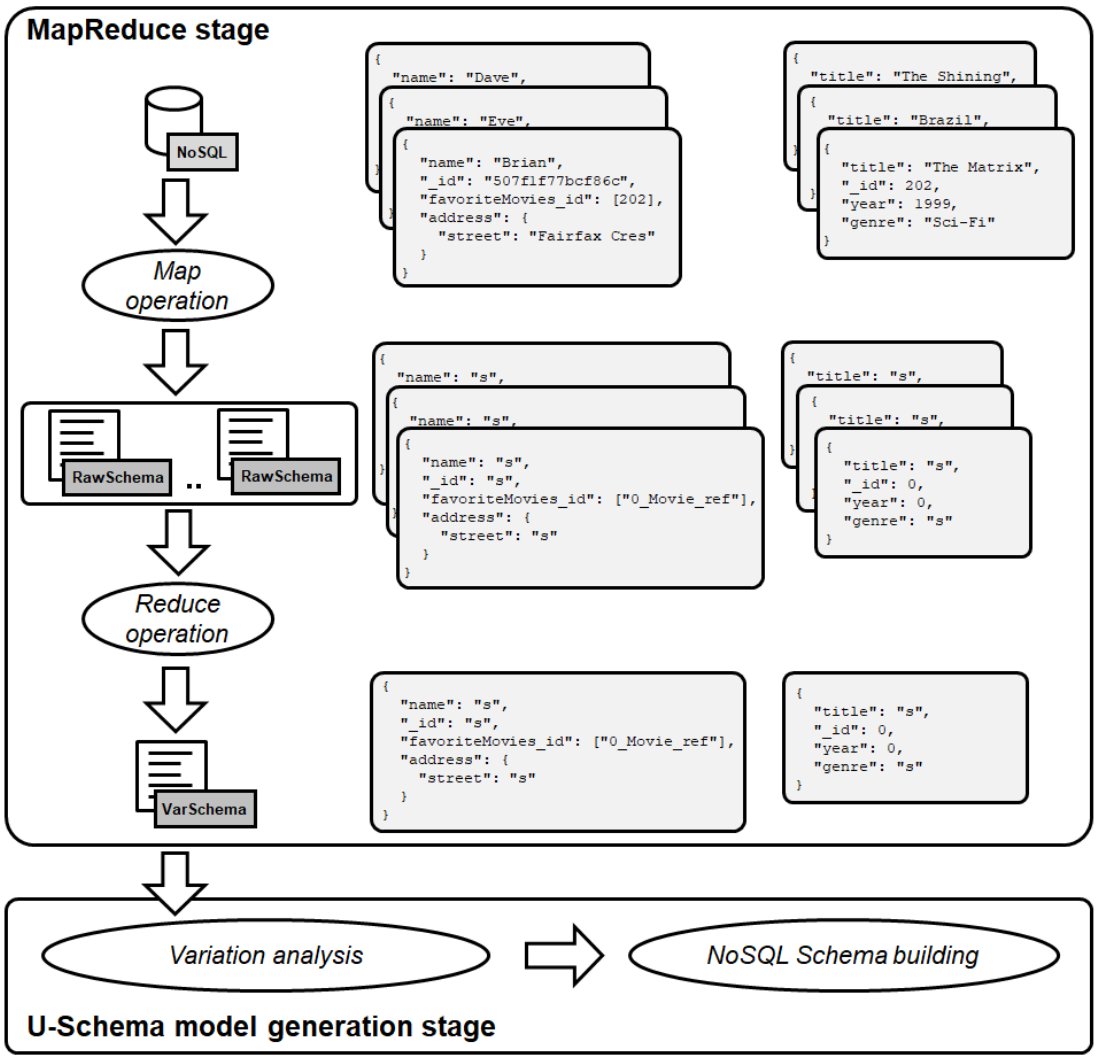

Figure 5.1: Generic Schema Extraction Strategy.

5.1.1 Inferring the Logical Schema

In the *map* operation, a *raw schema* is obtained for each object stored in the database. We call *raw schema* to an intermediate representation (JSON-like format) that describes the *data structure of a structural variation*: a set of pairs formed by the name of a property and

its data type. Given an object O stored in the database of an entity type e, its raw schema is obtained by applying the following 4 rules on the values of its properties p_i:

R1 Each value v_i of a p_i property is replaced by a value representing its type according to the rules R2 and R3.

R2 If v_i is of scalar or primitive type, it is replaced by a value that denotes the primitive type: `"s"` for String, `0` for numeric types, `true` for Boolean, and so on.

R3 If v_i is an embedded object, the rules R1, R2, and R3 are recursively applied on it.

R4 If v_i is an array of values or objects, rules R2 and R3 are applied to every element, and the array is replaced with an array of values representing types.

In the case of document systems, where the key is explicitly included in the documents, the representation of the structure will contain one scalar property with the name `_id`, representing the key of the entity type. Additionally, the following rule is applied to infer references between objects:

R5 Some commonly used conventions and heuristics are taken into account to identify references. For example, if a property name (with an optional prefix or suffix) matches the name of an existing entity type and the property values match the values of the `_id` property of such an entity. The value of the property is replaced concatenating the value indicated in rule R2 with the name of the entity type and the suffix `_ref`.

The process is repeated to obtain the *raw schemas* of the relationship types in the case of graph databases.

Figure 5.1 shows how the above rules are applied to *User* and *Movie* objects of a document store. A raw schema is obtained for each *User* object with identical structure, and the same for *Movie* objects.

Once the map function is performed, the reduce function collects all the identical raw schemas and outputs a single representative raw schema for each structural variation of an entity type, to which we will refer, hereafter, as *variation schema*. Figure 5.1 shows the

variation schemas obtained for *User* and *Movie* objects. Note that a *variation schema* will be generated for each structural variation of the objects.

In the case of graph and key-value systems, a preliminary stage is needed to achieve an efficient MapReduce processing, as explained in Sections 5.2.2 and 5.4.2.

We decided to build U-Schema models directly from the intermediate representation of the MapReduce output instead of building specific metamodels for each paradigm, because U-Schema already contains the abstractions present in each of the individual data models, and the transformation would have been redundant.

5.1.2 Generating a U-Schema Model

In the second stage, *variation schemas* are analyzed to build the U-Schema model. For this, a parsing process is connected to a schema construction process by applying the Builder pattern [58]. Variation schemas are parsed to identify its constituent parts: properties and relationships, as well as the entity type (or relationship type) to which they belong. Whenever the parser recognizes a part, it passes it to a builder that is in charge of creating the schema. A builder has been implemented for each data model, which captures how parts are mapped to U-Schema. The same parser is used for all the data models as its input are variation schemas. In the case of relational databases, only this second stage is needed, as schemas are already declared.

5.2 Graph Model Schema to U-Schema Models

Each element of the graph model defined above has a natural mapping to a U-Schema element, with the exception of relationship types, that map to two U-Schema elements: `RelationshipType` and `Reference`. The former represents a type or classifier whose instances are relationships between a pair of nodes, and can have variations based in their set of attributes, while the latter denotes a particular link between two nodes. Note that `Aggregation` and `Key` U-Schema elements do not have a direct correspondence to elements of the graph model. Next, we express the set of rules that defines the Graph to U-Schema canonical mapping.

R1. A graph schema *G* corresponds to an instance *uS* of the `uSchemaModel` metaclass of

`U-Schema` (i.e., a schema or model) with the same name:

$$uS \leftrightarrow G \parallel \{uS.name = name(G)\}$$

R2. Each different single-label entity type e that exists in G maps to a root `EntityType` et in the uS schema, whose name is that of the label associated to e:

$$et \leftrightarrow e \| \{et.name = name(e),\ et.root \leftarrow true\}$$

`EntityType` instances are included in the $uS.entities$ collection.

R3. Each different multiple-label entity type e that exists in G maps to a root `EntityType` et in the uS schema whose name is formed by concatenating the names of the set of $n > 1$ labels $L = \{l_1, \ldots, l_n\}$, and et inherits from each entity type $e_1, \ldots, e_n$ that corresponds to labels in L:

$$\begin{aligned} et \leftrightarrow e \parallel \{&et.name = concat(L),\\ &et.root \leftarrow true,\\ &et.parents = set\{map(e_1), \ldots, map(e_n)\}\} \end{aligned}$$

R4. Each relationship type r that exists in G maps to a `RelationshipType` rt and a `Reference` rf in the uS schema, which are named the same as the label associated to r.

$$r \leftrightarrow rt \parallel \{rt.name = name(r)\},$$

$$r \leftrightarrow rf \parallel \{rf.name = name(r)\}$$

`RelationshipType` instances are included in the $uS.relationships$ collection, and Rule R7 specifies how references are connected to other elements of the U-Schema schema.

R5. Each `variation schema` v of an entity or relationship type in G maps to a `Structural Variation` sv in the uS schema, which is identified by means of a unique identifier *index* (an integer ranging from 1 to $|EV|$ or $|RV|$). Structural variations are included in the collection *variations* that both entity types and relationship types have in a U-Schema schema.

R6. Let P^v be the set of properties of a `variation schema` v which maps to a `Structural Variation` sv. Each property $p_i^v \in P^v$ will map to an `Attribute` at_i^{sv} with the same name,

which is included in the collection *sv.features*. The type of the property will map to one of the types defined in the `Type` hierarchy defined in U-Schema, and a mapping has to be specified for each graph store. The property mapping can be expressed as:

$$p_i^v \leftrightarrow at_i^{sv} \parallel \{at_i^{sv}.name = name(p_i^v), \\ at_i^{sv}.type \leftrightarrow type(p_i^v)\}$$

R7. Each reference in a U-Schema schema *uS* has to be connected to other elements of *uS*. Let *rf* be a `Reference` which maps to a relationship type *r* according to Rule **R4**,

i) *rf* must be linked to the `EntityType` that maps to the entity type that denotes the destination nodes for the relationship *r*: $rf.refs\_to \leftarrow map(destination(r))$.

ii) Let *oe* the `EntityType` of *uS* that maps to the origin entity type of a relationship type *r* in *G* ($oe \leftrightarrow map(origin(r))$), *rf* will be present in the set of features of the variations of *oe* whose nodes are origin of edges that are instances of *r*.

iii) *rf* must be linked to the structural variation which features: $rf.isFeaturedBy \leftarrow sv$, where *sv* is the `StructuralVariation` that belongs to the relationship type that returns $map(r, RelationshipType)$.

iv) The `lowerbound` cardinality of *rf* would be 1 ($rf.lowerBound \leftarrow 1$) and the `upperbound` cardinality could be 1 ($ref.upperBound \leftarrow 1$) or ∞ ($ref.upperBound \leftarrow \infty$) depending on whether the instances of *r* in the database (i.e. arcs of type *r*) have one or more destination nodes for a given origin node.

5.2.1 Reverse Mapping Completeness

The graph model does not include the `Key` and `Aggregate` elements. Next, we provide a possible mapping for these two concepts.

- **Key**. Remember that the `Key` concept in U-Schema refers to those attributes that act as an object key or the set of attributes that form part of a reference to another object. As references between objects (nodes) in graphs are explicit in arcs, there is no need to include key information into the graph schema. However, that information could

be included in the nodes, for example, using a special *_keys* property holding the set of properties that act as key.

- **Aggregate**. An `Aggregate` *ag* that belongs to a particular `StructuralVariation` *sv*, where *ag.aggregates* is the aggregated variation *av*, could be mapped to a relationship type whose name is *ag.name* adding the prefix `AGGR_`, its origin entity type being *sv.container*, and its destination entity type being *av.container*. Origin and destination entity types should be created if they do not exist in the graph schema. Also, properties of *av* should be mapped using rules R2 to R7, as well as this rule (if an aggregate is part of the properties of *av*). Figure 5.1 shows an example JSON document of an U-Schema entity type `Person` that aggregates an object of the entity type `Address`. Figure 5.2 illustrates the reverse mapping where a relationship type named `AGGR_address_address1` connects a `Person` and `Address` nodes (we suppose that the aggregated variation identifier is 1.)

```
Person:
{
  name: "Diego",
  address: {
    street: "Espinardo Campus",
    number: 2
  }
}
```

Listing 5.1: Person with Address Aggregate.

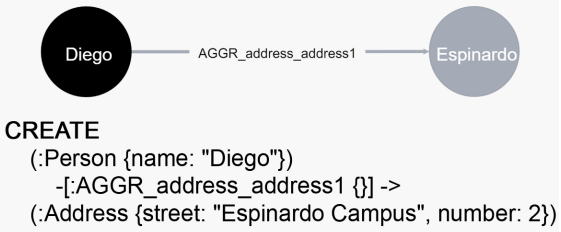

Figure 5.2: Person Aggregates Address in Neo4j.

5.2.2 Implementation and Validation of the Forward Mapping for Neo4j

A slightly revised strategy to that described in Section 5.1 has been applied to implement the forward canonical mapping for Neo4j. We chose this store because it is the most popular graph database.* It is schemaless and fits into the *labeled property graph data model*.

The strategy had to be revised because graph databases usually do not offer facilities to efficiently process the whole graph, and sometimes they even fail because of lack of resources. So we devised a preliminary stage that serialized the graph obtaining all the

*DB-Engines Ranking https://db-engines.com/en/ranking, (May, 2022).

nodes along with their outgoing relationships. Of each arch, the data included the source node with its properties, the properties of the arc, and the ID of the destination node. We modified the *map* operation of the generic strategy to construct all the raw schemas for nodes and edges with this serialization format. The serialization was organized in batches by using Spark Neo4j connector [109]. This way, an efficient schema extraction process was achieved. The reduce operation did not need any modification from that described in the generic strategy, generating variation schemas for both entity and relationship types from nodes and arcs, respectively.

The process finalizes with creating the U-Schema model by applying the mapping rules to the previous output (i.e. the *logical graph model*). The resulting schema for the User Profiles running example is shown in Figure 5.3. We also show the *union schema* in Figure 5.4.

The two experiments introduced in Section 5.7 were successfully carried out on the Neo4j database created for the running example and a *Movies* dataset available at the Neo4j website.[†]

Regarding scalability and efficiency of the model creation process, Table 5.2 show that the relative times with the reference query decrease as the size of the database increases. Neo4j, jointly with MongoDB show the worst ratio cases. This is because the query (average of watched movies by user) is, by chance, easily optimized by the database. In any case, as the database grows, the factor is never beyond 10x.

5.3 Document Model Schema to U-Schema Models

5.3.1 Canonical Mapping between Document Model and U-Schema

Each element of the document data model defined above has a natural mapping to a U-Schema element. Next, we present the rules for the canonical mapping.

R1. A document schema D corresponds to an instance uS of the `uSchemaModel` metaclass of `U-Schema` (i.e., a schema or model) with the same name:

$$uS \leftrightarrow D \parallel \{uS.name = name(D)\}$$

[†]Movie database, Neo4j website: https://neo4j.com/developer/example-data/.

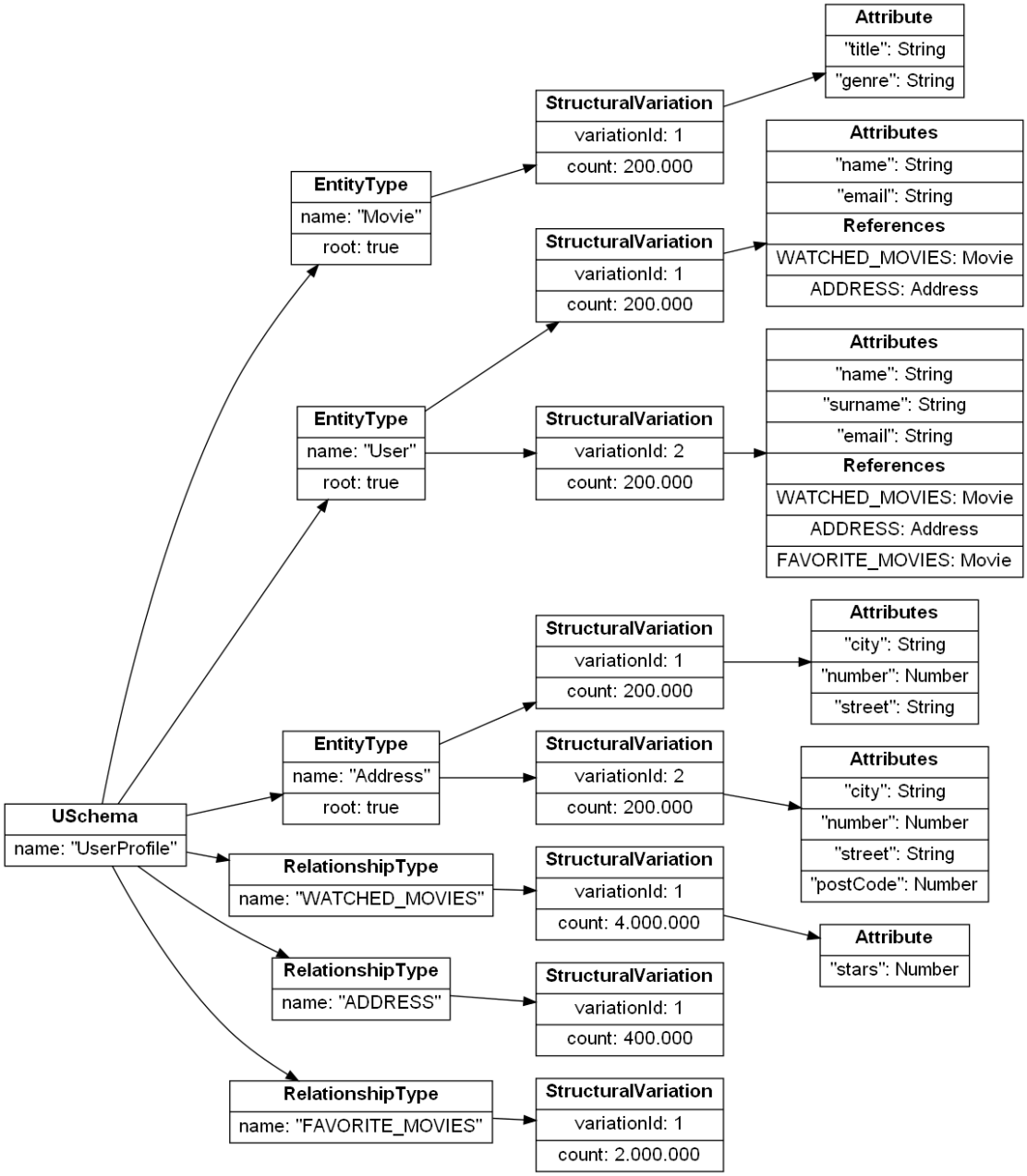

Figure 5.3: User Profiles Complete Schema for Graph Stores.

R2. Each entity type e that exists in D maps to a root `EntityType` et with the same name:

$$et \leftrightarrow e \| \{et.name = name(e),\ et.root \leftarrow true\}$$

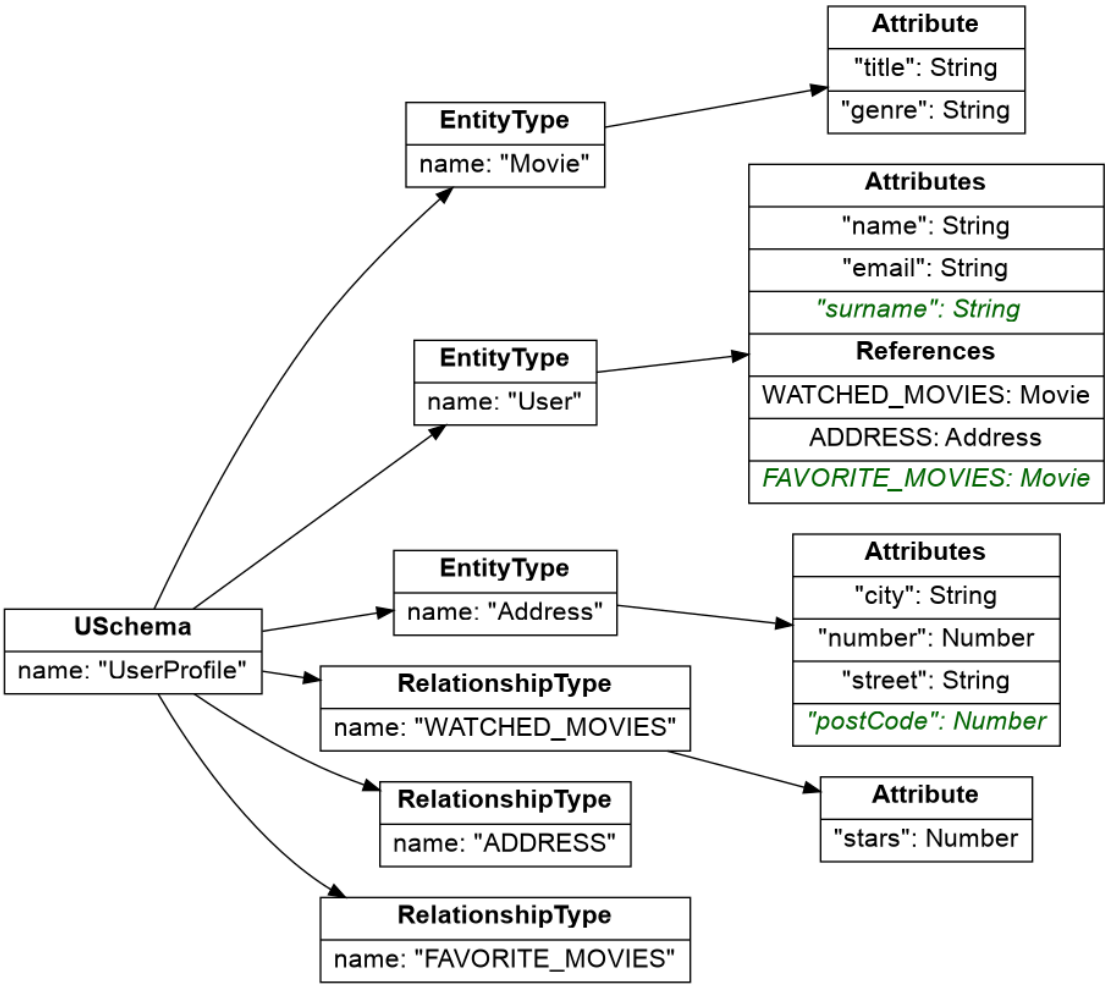

Figure 5.4: User Profiles Union Schema for Graph Stores.

uS.entities holds the set of instances of `EntityType`.

R3. Each variation schema v of e corresponds to a `StructuralVariation` sv of et in the uS schema, which is identified by means of a unique identifier *index* (an integer ranging from 1 to $|EV|$). Each property p_i^v of v will be mapped according to rules R4–R6.

$$sv \leftrightarrow v \parallel \{sv.variationId = idgen(), \quad sv.features \leftrightarrow properties(v)\}$$

`StructuralVariation` instances are included in the collection *et.variations*.

R4. If p_i^v is an attribute,

i) it will map to an `Attribute` at_i^{sv} with the same name, which is included in the collection *sv.features*. The mapping is the same as that defined in Rule **R6** of the mapping between the graph model and U-Schema.

ii) Additionally, if the attribute is the key of the entity type, a `Key` instance also exists in *sv.features* and is connected to the corresponding attribute at_i^{sv}.

R5. If p_i^v is an aggregate that has a set of n properties $G^v = \{g_i^v\}, i = 1..n$, it will map to three elements in the U-Schema model:

i) A non-root `EntityType` *nr* with the same name but capitalized and stemmed (function $name^*()$), which is included in the collection *uS.entities*:

$$nr \leftrightarrow p_i^v \parallel \{nr.name = name^*(p_i^v),\ nr.root \leftarrow false\}$$

ii) A `StructuralVariation` instance *av* included in *nr.variations*, and each property g_i^v is mapped recursively according to rules R4 to R6:

$$av \leftrightarrow p_i^v \parallel \{av.features \leftrightarrow G^v\}$$

iii) An `Aggregate` *ag* with the same name as the property, which is included in *sv.features*. This aggregate *ag* is connected to the structural variation *av* that it aggregates. The mapping would be:

$$ag \leftrightarrow p_i^v \parallel \{ag.name = name(p_i^v),\ av \in ag.aggregates\}$$

The cardinality of *ag* would be established as indicated in Rule R7-ii of the mapping between graph models and U-Schema models.

R6. If p_i^v is a reference, it corresponds to two elements of the U-Schema model:

i) A `Reference` *rf* with the same name, which is included in *sv.features*. The mapping is the same defined in Rule R7 of the mapping between graph models and U-Schema models.

ii) An `Attribute` *at* according to the mapping expressed in Rule R4-i, and *at* and *rf* appear connected in the *uS* schema: *at* exists in *rf.attributes* and *rf* is part of *at.references*.

5.3.2 Reverse Mapping Completeness

The only element of U-Schema that is not directly supported by the document model is the `RelationshipType`. `RelationshipType`s have structural variations, and some `Reference`s can specify (via its `isFeaturedBy` property) to which `StructuralVariation` of a `RelationshipType` they belong.

Given a `RelationshipType` *rt* in a U-Schema model, the reverse mapping for documents would map to an entity type *e* whose name is *rt.name* + *_REF*. Each variation of *rt* will correspond to a variation in *e*, applying mapping rule R3 (i.e., each `Attribute` in *rt* maps to an attribute of the corresponding variation of *e*). A reference property *p* will exist in all the variations of *e* that will map with rule R6. Then, each `Reference` *rf* that belongs to a `StructuralVariation` *v* of the entity type *et* to which $origin(rt)$ maps, where *ro* is the relationship type such that $ro = map(rt)$, and whose `isFeaturedBy` is a variation *vt* in *rt.variations*, will map to a reference property *r* named $name(e)$ + *_ref* by applying rule R6.

$$
\begin{aligned}
rf \leftrightarrow r \| \{ & rf \in v.features, \\
& et \leftrightarrow origin(rt), \\
& v \in map(et).variations, \\
& vt = rf.isFeaturedBy, \\
& vt \in rt.variations, \\
& r.name \leftarrow name(e) + \_ref \}
\end{aligned}
$$

Figure 5.5 illustrates the application of the reverse mapping explained above for a U-Schema model containing a `RelationshipType` for the *watchedMovies* relationship type of the running example. It can be appreciated how the document schema would contain an entity type named *WatchedMovie_REF*, which has a structural variation for the single `StructuralVariation` of the `RelationshipType` that exists in the U-Schema schema. That variation is connected to the attributes named *stars* and *movie_id*. Also, there exists a reference to the entity type *Movie*, and a reference and attribute named *watchedMovie_REF* are present

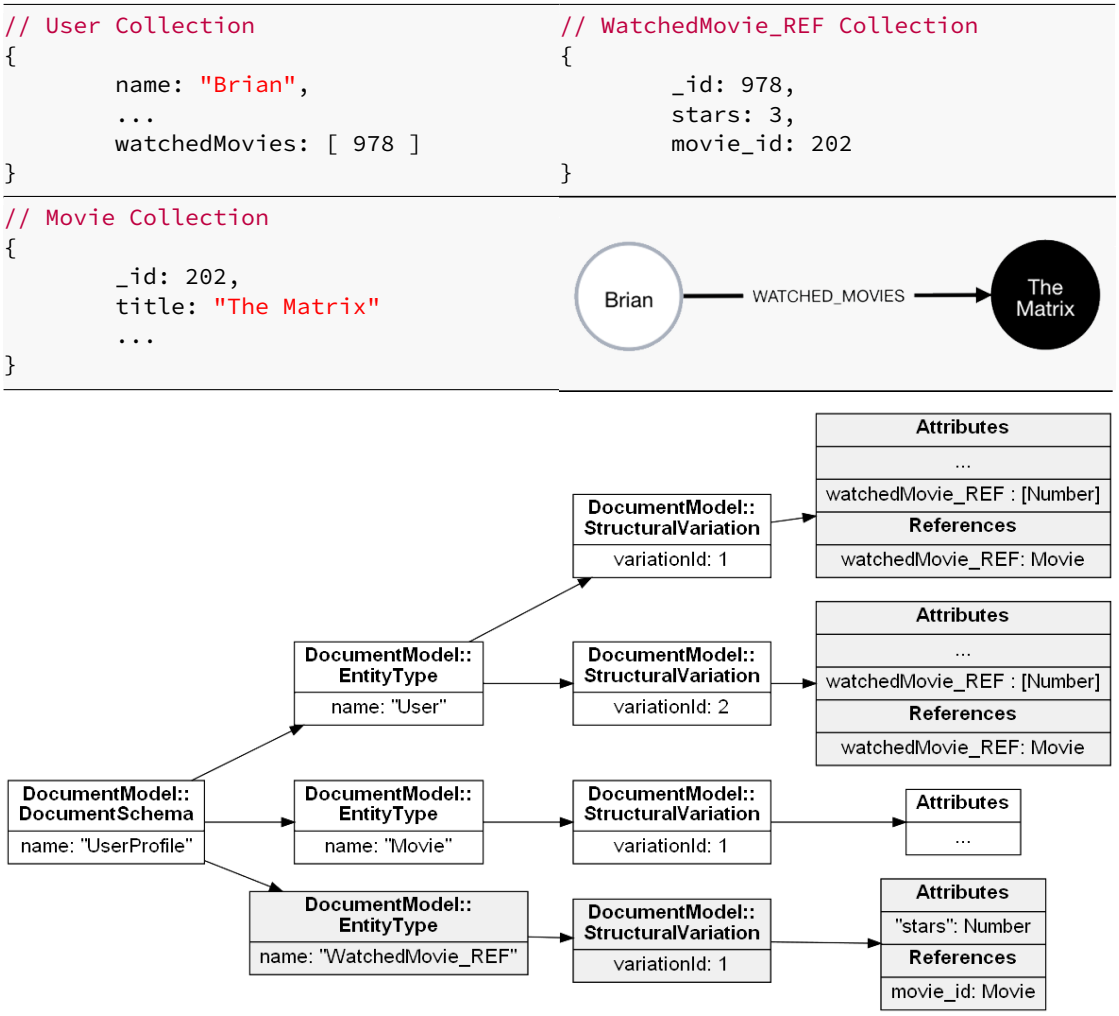

Figure 5.5: Example of Application of the Reverse Mapping from a `RelationshipType` of U-Schema to a Document Schema.

in the structural variations of the origin entity type (*User* in our example). The reference will connect the *User* objects with the *WatchedMovie_REF* objects.

Some document systems provide the *dbref* construct to record references between documents, which can include fields. In these systems, the document data model shown in Figure 2.5 could be extended to consider that references can have attributes. Then, the document model would include all the U-Schema elements, as it would also support relationship types.

5.3.3 Implementation and Validation of the Forward Mapping for MongoDB

Once the output of the MapReduce described in Section 5.1 is produced, i.e., the set of variation schemas, the generation of the U-Schema model is achieved by following the mapping rules described above. The only remarkable aspect is that while the root entity types are discovered by the MapReduce process, aggregated entity types reside unfolded inside the variation schemas. It is needed to recursively process all the aggregated objects to build the non-root `EntityType`s and match the properties to identify the `StructuralVariation`s.

The schema that would be inferred for the running example is shown in Figure 5.6 and the union schema in Figure 5.7.

The common validation strategy of Section 5.7 was successfully applied in MongoDB, with a database created for the running example, and with the *EveryPolitician* dataset[‡].

As with Neo4j, MongoDB shows worse ratio cases than with other two database implementations, as shown in Table 5.2. Again, this may be caused by the chance that the query benefits by some optimizations built in the database. The ratio also goes down as the size of the database doubles, with the exception of the Small and Medium times, that are similar (17.58x and 17.71x). The ratio then goes down from around 18x to 10x for the biggest case.

5.4 Key-Value Model Schema to U-Schema Models

The mapping between U-Schema and Document model would be applicable for K/V, the only exception being that Rule R4-ii should be removed, and a new rule has to be added because the notion of key is different in this data model.

R7. Each `StructuralVariation` *sv* in the U-Schema model contains a `Key` instance *k* in *sv.features* whose value of *k.name* is *_id*, and it is not connected to any `Attribute`.

5.4.1 Reverse Mapping Completeness

The same reverse completeness mapping rules exposed in Section 5.3.2 for the document model are applicable in this case.

[‡]Available at http://docs.everypolitician.org/.

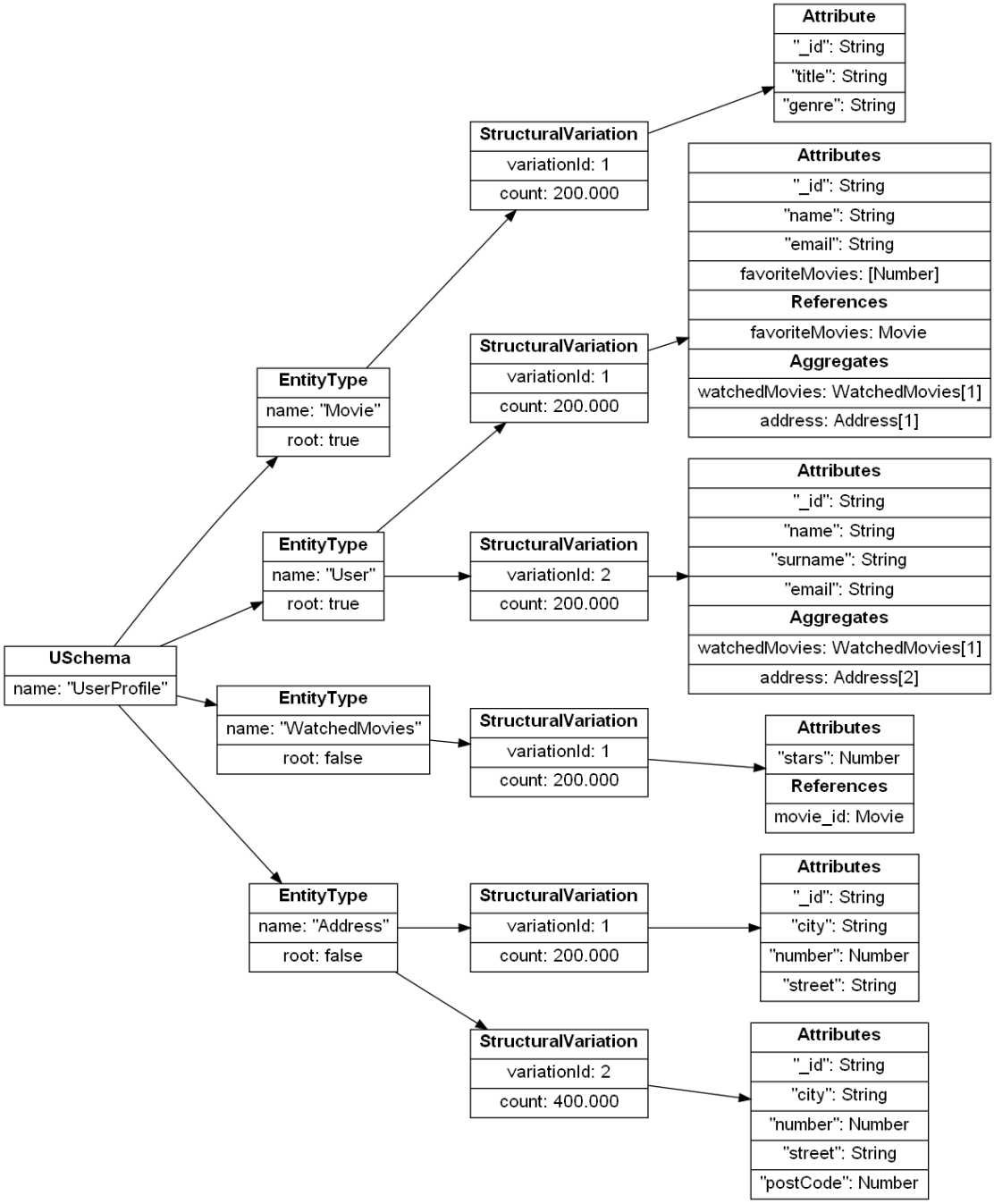

Figure 5.6: User Profiles Complete Schema for Document Stores.

5.4.2 Implementation and Validation of the Forward Mapping for Redis

Redis has been used for the implementation and validation of the general strategy applied for key-value stores. Redis is the most popular key-value database[§].

[§]As shown in https://db-engines.com/en/ranking. Redis appears in the 6nd position (May, 2022).

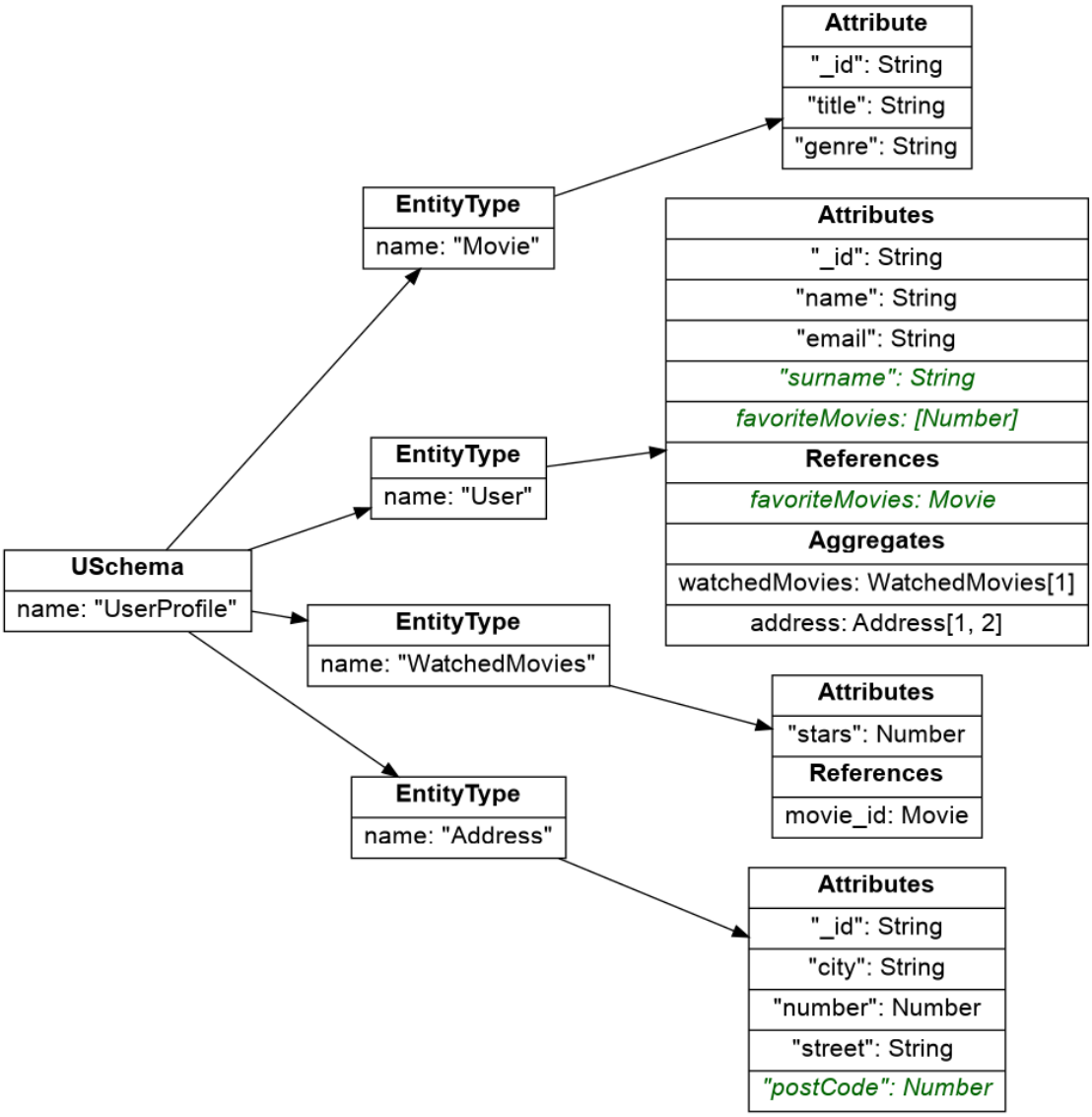

Figure 5.7: User Profiles Union Schema for Document Stores.

A preliminary stage is performed to join all the properties of each entity variation. To do this, a simple MapReduce operation is performed over the database assuming that properties are encoded using the *flattened object-key pattern*. Spark [109] was used to implement this stage. First, every database pair is mapped to a new pair whose key is the name of the entity type along with its identifier, and the value is formed by the property's name and its type. Then, the *reduce* operation joins all pairs of objects that belong to the same object.

The result is a set of JSON objects that are similar to those stored in a document database. Now, the two stages of the common strategy are performed: a MapReduce processing to obtain the set of variation schemas, followed by the generation of the U-Schema model, which is similar to the document model with the exception of the key generation using the rule R7.

The schema extracted for the running example is the same as for documents, shown in Figure 5.6. The union schema is shown in Figure 5.7.

The schema extraction process was validated using a database built for the running example, as well as using a real-world dataset. The same *EveryPolitician* dataset used with MongoDB (Section 5.3.3) was inserted into Redis.

The performance of the Redis schema inference process implementation versus the query gets better than in MongoDB or Neo4j. This is because the query itself has to process the whole database, as Redis does not include a query language. Note also that in absolute times, the Redis implementation is the slowest, which confirms that the calculation of an aggregate value is not an appropriate operation for a K/V store. As in previous implementations, the ratio goes down from 11.46x to 5.76x as the database doubles.

5.5 Columnar Model Schema to U-Schema Models

In the case of columnar stores, the canonical mapping would be the same as the one defined for document stores. Relationship type would be the only element of U-Schema not included in the columnar model.

5.5.1 Reverse Mapping Completeness

The data model for columnar databases includes the same abstractions than those established for the document data model. Thus, the reverse mapping rules are the same to those introduced in Section 5.3.2. Only relationship types do not have a direct mapping to the model, and the same approach used in documents can be implemented: the new entity type with a name convention to hold the structure residing in the references, and the reference itself on the origin entity type variations.

5.5.2 Implementation and Validation of the Forward Mapping for HBase and Cassandra

We implemented the forward mapping for Hbase and Cassandra. In the case of HBase, we applied the common strategy of Section 5.1, with the MapReduce operation identifying the default and aggregated column families, and building the variation schemas. In the case of Cassandra, the API was used to retrieve the database schema, and then build the U-Schema model.

Validation was carried out as described in Section 5.7. A database was created for the running example, and the same *EveryPolitician* real world dataset used in MongoDB and Redis, introduced in Section 5.3.3, was injected into Hbase and Cassandra.

Figures 5.6 and 5.7 show the variation schemas obtained for the running example, which are the same for all the aggregation-based stores.

As shown in Table 5.2 and Figure 5.8, Hbase shows the best performance of the inference regarding the ratio relative to the aggregated query. As with all systems, with a slight difference in the two bigger databases (1.8x to 2.04x), the ratio decreases as the size of the database increases. This confirm the scalability of the schema extraction approach. HBase, like Redis, is specialized in fast random-access queries, but the aggregated query has to process most of the database, making the times very close to the full process of the database performed in the schema extraction. Thus, ratios go from just 6x slower to around 2x slower in the case of the Larger databases.

The performance of building the Cassandra model was not recorded as no inference process is required because the schema is already declared.

5.6 Relational Model Schema to U-Schema Models

The relational model is completely integrated in U-Schema, but the latter has the `Aggregate` element which is not present in relational schemas. Moreover, all the tuples have the same structure, so that the number of structural variations for an entity type is limited to one. Next, we expose a set of rules that specify the canonical mapping between relational and U-Schema models. We will use the terminology of table data models. **R1**. A relational schema D corresponds to a `uSchemaModel` instance uS in U-Schema (i.e., a U-Schema

model) with the same name:

$$uS \leftrightarrow R \parallel \{uS.name = name(R)\}$$

R2. Each table t in R representing an entity type maps to two elements of uS: a root `EntityType` et with the same name, and a `StructuralVariation` sv that represents the only structure of the table that exists in the database. An identifier is generated for the variation sv and its features are mapped to the columns of the table t by applying the rules R4 to R6. This mapping can be expressed as follows:

$$et \leftrightarrow t \parallel \{et.name = name(t),\ et.root \leftarrow true\},$$
$$sv \leftrightarrow t \parallel \{sv.id \leftarrow idgen(),\ sv.features \leftrightarrow columns(t)\}$$

`EntityType` instances are included in $uS.entities$ and sv is included in $et.variations$.

R3. Each table r in R representing a relationship type maps to two elements of uS: a `RelationshipType` rt with the same name, and a `StructuralVariation` sv that represents the only structure of the table that exists in the database. The mapping between r and sv is solved as in rule R2.

$$rt \leftrightarrow r \parallel \{rt.name = name(r)\},$$
$$sv \leftrightarrow r \parallel \{sv.id \leftarrow idgen(),\ sv.features \leftrightarrow columns(r)\}$$

`RelationshipType` instances are included in $uS.relationships$ and sv is included in $et.variations$.

R4. Each column c of a table t is mapped to an `Attribute` at with the same name, and the data type of the column will map to one of types defined in the `Type` hierarchy of U-Schema (a mapping between types has to be specified for each relational system.) The mapping can be expressed as follows:

$$at \leftrightarrow c \parallel \{at.name = name(c),\ at.type \leftrightarrow type(c)\}$$

Attributes of an `EntityType` et are included in the collection $sv.features$, where sv is the only structural variation that et has.

R5. The primary key pk of a table t is mapped to a `Key` k and the collection $k.attributes$

includes the attributes that maps to the columns that form *pk*. The name of *k* is the name of the attribute (if there is just one), or *t.name* + *_pk* otherwise (*pkname()* function):

$$pk \leftrightarrow k \,||\, \{k.name = pkname(pk), \quad k.attributes \leftrightarrow columns(pk)\}$$

For each attribute $at \in k.attributes$, $at.key = k$. *k* is also included in *sv.features*.

R6. Each foreign key *fk* of a table *t* to a table $s = target(fk)$ is mapped to a `Reference` *rf*, and the collection *rf.attributes* includes the attributes that map to the columns that form *fk*. The name of *fk* is the name of the attribute (if there is just one), or *s.name* + *_fk* otherwise (*fkName()* function). The reference *rf* is included in *sv*. It also refers to the entity type that maps to the target table *s*:

$$fk \leftrightarrow rf \,||\, \{rf.name = fkName(fk), \quad rf.attributes \leftrightarrow columns(fk), rf.refsTo = map(s)\}$$

5.6.1 Reverse Mapping Completeness

The U-Schema elements that are not present in the relational model are `Aggregate`, and (multiple) `StructuralVariation`. Next, we describe some possible mappings for these elements.

- The canonical mapping only takes into account a `StructuralVariation` per schema type (resp. table). If an schema type has several `StructuralVariation`s, then two possible alternatives are: (i) mapping each variation to a table with a distinctive naming scheme, and (ii) mapping all variations to a single table where the columns result of the union of the set of properties of each structural variation. In the latter case, the tuples of the table will have `NULL` values in the columns not corresponding to their structural variation. Obtaining the different entity variations from a table would require the analysis of all the tuples to register all the different set of non-NULL columns. This could be carried out with a similar operation to the MapReduce described in the common strategy of Section 5.1.

- Each `Aggregate` *ag* in an `StructuralVariation` *sv* of a given `EntityType` *et* could be mapped to elements of the relational model also in several ways: (i) an additional

table *t* with the name of the aggregate *ag.name* and the columns mapped to properties in the `StructuralVariation` *ag.aggregates* using rules R4 to R6. A foreign key column is added to *t*, and a primary key to the table mapped to *et*. (ii) If the aggregate cardinality is one-to-one, the attributes of the *ag.aggregates* variation could be incorporated into the table that maps to *et*. The aggregation relationship between *User* and *Address* in the running example schema has been mapped using the second alternative, as shown in Figure 2.9.

5.6.2 Implementing and Validating the Relational Schema Extraction Process

In the case of relational databases, it is not necessary to infer schemas: U-Schema models can be obtained from relational schema declarations. We chose MySQL to implement the set of rules exposed above for the relational to U-Schema mapping. Rule R3 cannot be applied as the schema does not distinguish between relationship and entity tables. This information could be provided, for example, through name conventions, which could also be used to specify aggregation tables.

The model generation process is straightforward, and it works following the described mapping rules. First, R1 is applied to create and name the model, then an `EntityType` and a `StructuralVariation` are created for each table (R2). An `Attribute` is created for each column of a table (R4). Next, `Key`s are created for primary keys in tables, which will have references to `Attribute`s that have been instantiated previously for the columns that are part of the primary key (R5). Finally, `Reference`s are created for foreign keys in tables, and each `Reference` will be connected to elements previously created according to the U-Schema metamodel (R6).

The validation has been performed on the *Sakila* database available at the MySQL official website[¶]. Sakila contains 16 tables, and the average numbers of columns and references between tables are, respectively, 5.6 and 1.4. The smallest table has 3 columns, and the biggest one 13 columns. We have checked the correction of the U-Schema model generation by comparing the model obtained with the information on the database available at the MySQL website (SQL creation files and official diagrams). In the study of performance

[¶]Sakila can be downloaded from `https://dev.mysql.com/doc/index-other.html`, and documentation is available at `https://dev.mysql.com/doc/sakila/en/sakila-structure.html`.

and scalability, as with Cassandra, relational databases have not been considered because schemas are already available.

5.7 Validation of the Schema Building Process

To validate our schema building process, we have applied the same validation for the four kinds of NoSQL paradigms. For each system, we used two databases, a synthetic one based on the running example, an a real dataset. In each one of them, two experiments were carried out: (i) a round-trip strategy to check that the obtained U-Schema model is equivalent to the schema used to synthesize the database or the schema of the real existing database; and (ii) two queries are issued on the real and synthesized databases to assure that at least a data object exists for each inferred structural variation (`all variations exist` query) and that the extraction process correctly calculates the number of data objects of each variation (`correctness count` query). In the case of the relational model, only the second experiment is performed, as only the canonical forward mapping must be implemented, because there is no need to infer the logical model of the database.

The round-trip experiment consisted in the following steps. First, we manually created a U-Schema model (i.e. a schema) with the desired database structure (or the existing structure in the case of the real database). The running example model covers all the elements that can be mapped into the logical data model of the corresponding paradigm, but this may not be the case for the real dataset. To populate the initial running example database, we randomly created elements according to the defined model. Afterwards, we inferred the implicit schema, and finally verified that this schema was equivalent to the original U-Schema model: the resulting model can differ in the ordering of the different variations found for each entity or relationship type, this is why in this case we could not use standard model comparison tools, so we built a custom U-Schema model compare utility.

To evaluate the scalability and performance of the U-Schema model building algorithms, we have generated four datasets of different size for the running example. The larger database contains 800,000 objects for the *User* and *Address* entities, 400,000 for *Movie*, and a mean of 20 watched movies and 20 favorite movies. *User* and *Address* have the same number of objects in each of their variations. The rest of datasets reduce the number of objects and relationships in a factor of 2, 4, and 8, as shown in Table 5.1.

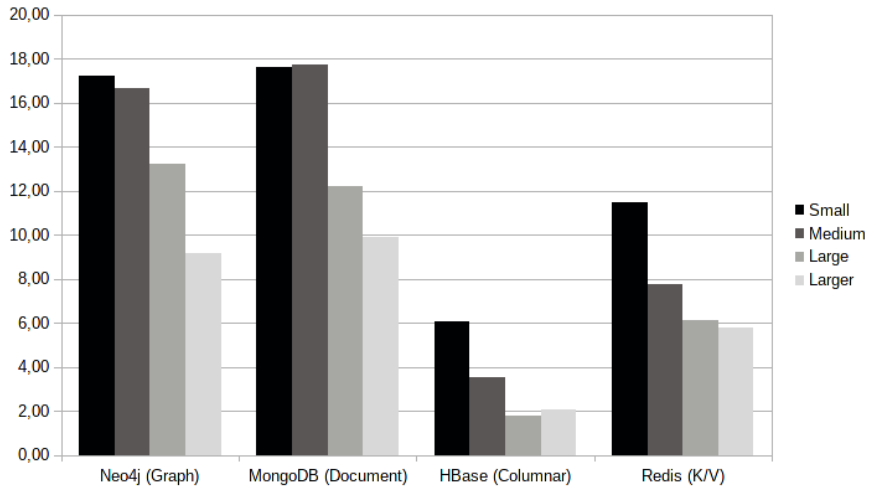

Figure 5.8: Inference to Query time ratio.

Size/Item	User	Movie	Watched/Favorite Movies	Nodes	Relationships
Larger	800k	400k	20/user	2,000k	24,800k
Large	400k	200k	10/user	1,000k	6,400k
Medium	200k	100k	5/user	500k	1,700k
Small	100k	50k	3/user	250k	550k

Table 5.1: Database Sizes.

All the performance tests were run on an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 48 GB of RAM and using SSD storage. To give a meaningful expression of the scalability of the schema inference process, instead of comparing absolute times, we used as a time baseline an aggregate query that calculates the average of watched movies by users. This query could be representative of those obtaining periodic reports, so we suppose that the database is not optimized for it. In this way, we can get results that are independent of the different configurations in the deployment. Table 5.2 show the different times for the queries, schema inference, and the normalized value (inference time divided by query time) for the database sizes in Table 5.1. Figure 5.8 shows a diagram with the normalized value for each kind of database. We expected the ratio to diminish as the size of the database increases, as the initialization time of the MapReduce framework becomes smaller with respect to the total inference time. Moreover, in all cases the ratio stays in the range of 17.58x (MongoDB, smaller case) to 2.04x (HBase, biggest case), and for the biggest case, the

DB		Small	Medium	Large	Larger
Neo4j	Query (ms)	686	1,213	3,165	12,016
	Inference (ms)	11,821	20,177	41,814	109,724
	Normalized	17.23	16.63	13.21	9.13
MongoDB	Query (ms)	295	380	840	2,366
	Inference (ms)	5,187	6,730	10,226	23,452
	Normalized	17.58	17.71	12.17	9.91
HBase	Query (ms)	931	1,942	6,419	24,023
	Inference (ms)	5,615	6,840	11,526	49,042
	Normalized	6.03	3.52	1.80	2.04
Redis	Qyery (ms)	1,002	2,833	10,091	43,888
	Inference (ms)	11,487	22,013	61,505	252,794
	Normalized	11.46	7.77	6.10	5.76

Table 5.2: Times for inference and queries for all the database implementations.

inference reaches a maximum of about 10x slower (MongoDB). This is expected as the query only has to process a part of the database while the inference treats the whole database.

With the extracted U-Schema model, we build a set of queries on the databases to perform the second experiment:

1. **All variations exist** The database must store, at least, a database object for each entity type variation (and relationship type variation in the case of a graph store) present in the extracted U-Schema model.

2. **Count correctness** No other variations are present in the database, i.e., the total number of objects in the database matches the sum of objects that belong to each structural variation of the entity types present in the extracted model (`count` attribute included in the `StructuralVariation` metaclass of the U-Schema metamodel.) Also, this check would be performed for relationship type variations in the case of graph stores.

6

Extracting U-Schema logical schemas from code of database applications

Strategies to discover schemas from stored data were addressed in Chapter 5.But schemas can also be extracted from code of applications accessing databases. In this chapter, a code analysis strategy that extracts U-Schema models will be presented. Throughout the chapter we will explain the model-driven reverse engineering process devised to implement that strategy: the chain of transformations, the involved metamodels, and the testing of each transformation. We have taken advantage of the extracted information in our reverse engineering process to tackle the automation of a refactoring, in particular, the removal of join queries to improve the performance. Finally, we will show the validation of the schema extraction and refactoring processes.

6.1 Overview of the approach

This section is devoted to outline the approach. A example of operation on a document store will be used to illustrate the information to be discovered in the code analysis to obtain the logical schema and automate the join removal refactoring.

In document stores, semi-structured objects are stored in collections, and each object

stored has a JSON-like structure: an object consists of a set of fields that are name and value pairs, where the value can be a primitive type value, an array of objects, or another embedded object. For example, Figure 6.1 shows *user* and *movie* objects of a document store that registers information about a streaming service, such as subscribed user data that includes personal data and a list of movies they have watched. User objects have the `watchedMovies` field that hold an array of embedded objects that have three fields: `_id` that records the identifier or key, `movie_id` that contains the identifier of the watched movie (i.e. a reference), and `starts` that holds the score given by the user to the watched movie.

```
// User Collection
{
  name: "Brian",
  surname: "Caldwell",
  email: "brian_caldwell@gmail.com",
  watchedMovies: [
    {
      stars: 7,
      movie_id: 202
    },
    {
      stars: 10,
      movie_id: 303
    }
  ]
}
```

```
// Movie Collection
{
  _id: 202,
  title: "The Matrix"
  director: "The Wachowskis"
},
{
  _id: 303,
  title: "The Godfather",
  director: "Francis Ford Coppola"
}
```

Figure 6.1: Users and Movies objects in the "streaming service" store.

```
1. user = db("Users").query(name == "Brian")
2. movie = db("Movies").query(
              _id == user.watchedMovies[0].movie_id)
3. if (user.watchedMovies[0].stars >= 5)
      println user.name+user.surname+user.email
      println "Last watched movie:"
      println movie.title+user.watchedMovies[0].stars
```

Listing 6.1: Pseudo-code of the FWM database operation.

Listing 6.1 shows an example of pseudo-code of an operation on the "User Profiles Running Example" store. We refer this script as "First-Watched-Movie" (FWM), and consists of three statements. First, a query selects an user by name, after a join query retrieves the first movie the selected user watched, and then if the user scored that movie with a number

of stars greater or equal than 5 the following information is printed in the console: the full name of the user, and the title and number of stars of the retrieved movie.

To ease the code analysis, a more abstract representation is obtained from the AST provided by a parser. We have defined a representation formed by a model that expresses the control flow, and whose nodes and edges are linked to the elements of another model that represents code at the level of an abstract syntax. First, the source code will be injected into a Code model, and the Control Flow model is obtained by analyzing the Code model, as described in Section 6.2. Extracting the logical schema and automating database refactorings requires to discover the data physical structure and the CRUD operations applied on the data. This information is captured in the DOS model, which is obtained by traversing the control flow model, as explained in Section 6.3. More specifically, the traversal should visit statements to find the following elements:

- *Data containers* (e.g., collections in document stores or tables in relational or columnar databases). They could be identified in the analysis of CRUD operations, e.g. the argument of calls `db()` in queries of the FWM example.

- *Structure of objects* stored in a particular container, i.e. a set of properties formed by a name and type pair. Properties are discovered from the object's fields. Fields can be found in expressions that use dot notation to access to fields of an object that a variable holds, e.g., `user.name` or `movie.title`.

- *Variables* holding database objects. They are discovered by analyzing expressions such as method invocations, assignments, and arguments. For instance, `user` and `movie` variables would be detected in the assignments of the statements 1 and 2 in the FWM example.

- *Types of the properties*. They can be primitive, collection (e.g. array or list), aggregate, or reference. Primitives types could be found from expressions such as assignments or conditions, e.g., `name == Brian`. An *array* type from expressions accessing to array elements by indexing, For instance, the analysis of `user.watchedMovie [0].movie_id` would identify that the `watchedMovies` property of users is an array. In that expression, it would also be detected that the elements of the array are objects that have the `movie_id` property. Therefore, the type of `watchedMovies` would

be an array of an aggregate type named as the field but singularizing. We will use the *non-root* entities to distinguish aggregate entities from those that correspond to containers, user and movie in the example.

- *CRUD operations*. They will be detected by identifying calls to operations of a particular API to manage databases. In the FWM example, assuming that the *query()* method is used to issue a read operation on the store indicated by the argument of the *db()* method, two Read operations would be detected in the statements 1 and 2.

- *Reference and Join queries*. When the condition of a query includes a equality check between the object identifier field and another field of a previously retrieved object, that query is identified as a join query. Thus, the field of the another object would a reference, and a reference type would detected for the corresponding property. In our example, the query on movies (statement 2) would a join query, and the type of the movie_id property would be a "Reference to Movie".

Figure 6.2 shows the information that the DOS model would have in the case of the pseudo-code of the FWM example. The data physical structure appears at the top of the figure, and the queries at the bottom. Note that the type of the fields `surname` and `email` of *User*, and `title` and `director` of *Movie* would not be identified from the pseudo-code, and the type String would be assigned by default, as they appear in print statements. In our approach, the data physical schema is transformed into a logical schema represented in the U-Schema generic metamodel, as explained in Section .

The DOS model may also be used to detect candidate database refactoring. We will illustrate this usefulness by addressing the join query removal refactoring. A join query involves four elements: a source container, a target container, a query on the source container, and the condition on the join query that selects the object of target container. Removing a join query is possible if the properties of the target entity are copied (duplicated) into the source entity. In this way, the query on the source container is enough to retrieve all the necessary information. Not all the properties of the target entity should be copied but those that are accessed in the code that follows the join query. In our example, the `title` field should be copied into *WatchedMovie* objects but not the `director` field, as the statement that follows the query contains the `movie.title` expression but not `movie.director`.

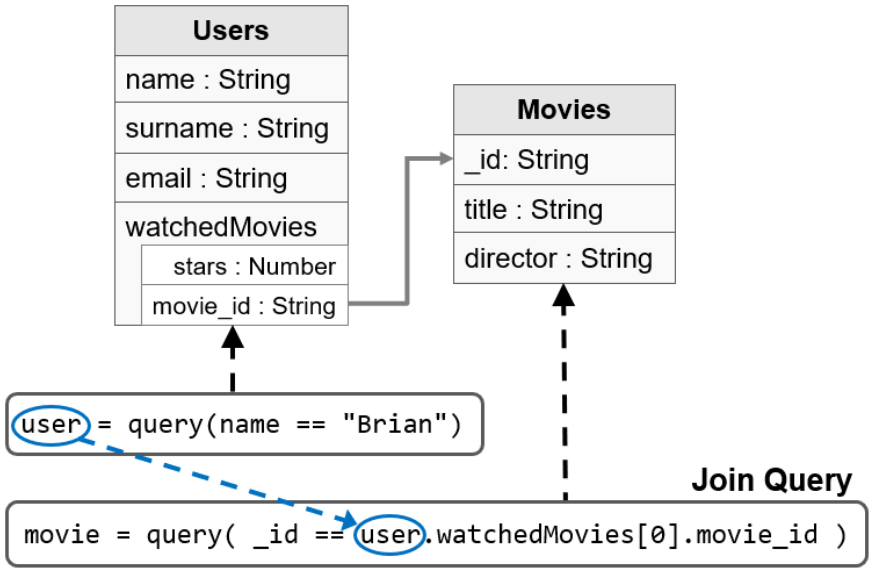

Figure 6.2: Entities and Queries extracted for the FWM example.

Therefore, a code analysis is here proposed to discover the data to be duplicated for each join query, as described in Section 6.4. For achieving this, the list of join queries is iterated, and for each of them the statements that follow it are inspected to discover the fields to be copied. This analysis provide to database administrators additional information that can help them to select what join queries should be eliminated such as: the source and target container, the join query, the original and modified query on the source container, number of lines where the retrieved data is used, and other queries on the two containers involved in the join in order to the administrator can check how often the duplicated data are updated. All the information obtained in this code analysis is called a "join query removal plan."

Like any schema change operation, the data duplication associated to a join query removal refactoring would entail to update the schema, database, and application code. We have automated this refactoring as follows: (i) the logical schema is changed to add duplicated attributes from the referenced entity to the referencing entity, (ii) the database is updated by adding the duplicated fields to all the referencing objects, and (iii) the code is rewritten to remove the join query, and change statements accessing to duplicated fields to directly access to the object with data duplicated. In the case of the FWM script, (i) the *title* attribute would be added to the *WatchedMovie* entity type in the schema, this is imple-

mented as an operation on the U-Schema model. (ii) the `watchedMovies` array of each *user* object is updated, so that each existing object is replaced by another one having the `title` field of the referenced movie, and (iii) the FWM script would be rewritten as shown in Listing 6.2.

```
1. user = db("Users").query(name == "Brian")
2. if (user.watchedMovies[0].stars >= 5)
    println user.name + user.surname + user.email
    println "Last watched movie":
    println user.watchedMovies[0].movie_title
            + user.watchedMovies[0].stars
```

Listing 6.2: Pseudo-code updated when join query is removed.

Figure 6.3 shows the sequence of stages of the strategy outlined above. Code is injected in a Code model from which a Control Flow model is obtained. This model is analyzed to generate the DOS model, which is the input to two processes: the transformation that generates the logical schema and the analyzer that generates join query removal plans. These plans are provided to administrators who choose what plans should be applied. Finally, schema, database and code are updated for each selected plan.

```
1  const MongoClient = require("mongodb").MongoClient;
2
3  const url = "mongodb://modelum.es/db:27017";
4  const dbName = "streamingservice";
5
6  const client = new MongoClient(url);
7  client.connect(err => {
8
9   client.db(dbName).collection("users").findOne(
10   { name: "Brian" }, (err, user) => {
11   client.db(dbName).collection("movies").findOne(
12    { _id: user.watchedMovies[0].movie_id },
13    (err, movie) => {
14      if (user.watchedMovies[0].stars >= 5) {
15       console.log(user.name + " " + user.surname);
16       console.log(user.email + " Last watched movie:");
17       console.log(movie.title + " " + user.watchedMovies[0].stars);
18      }
19    });
20  });
21 });
```

Listing 6.3: JavaScript code for the pseudo-code in Listing 6.1 (data stored in MongoDB).

Listing 6.3 shows the FWM pseudo-code is expressed in JavaScript in order to be used as running example in the following sections. We will suppose that MongoDB is the ac-

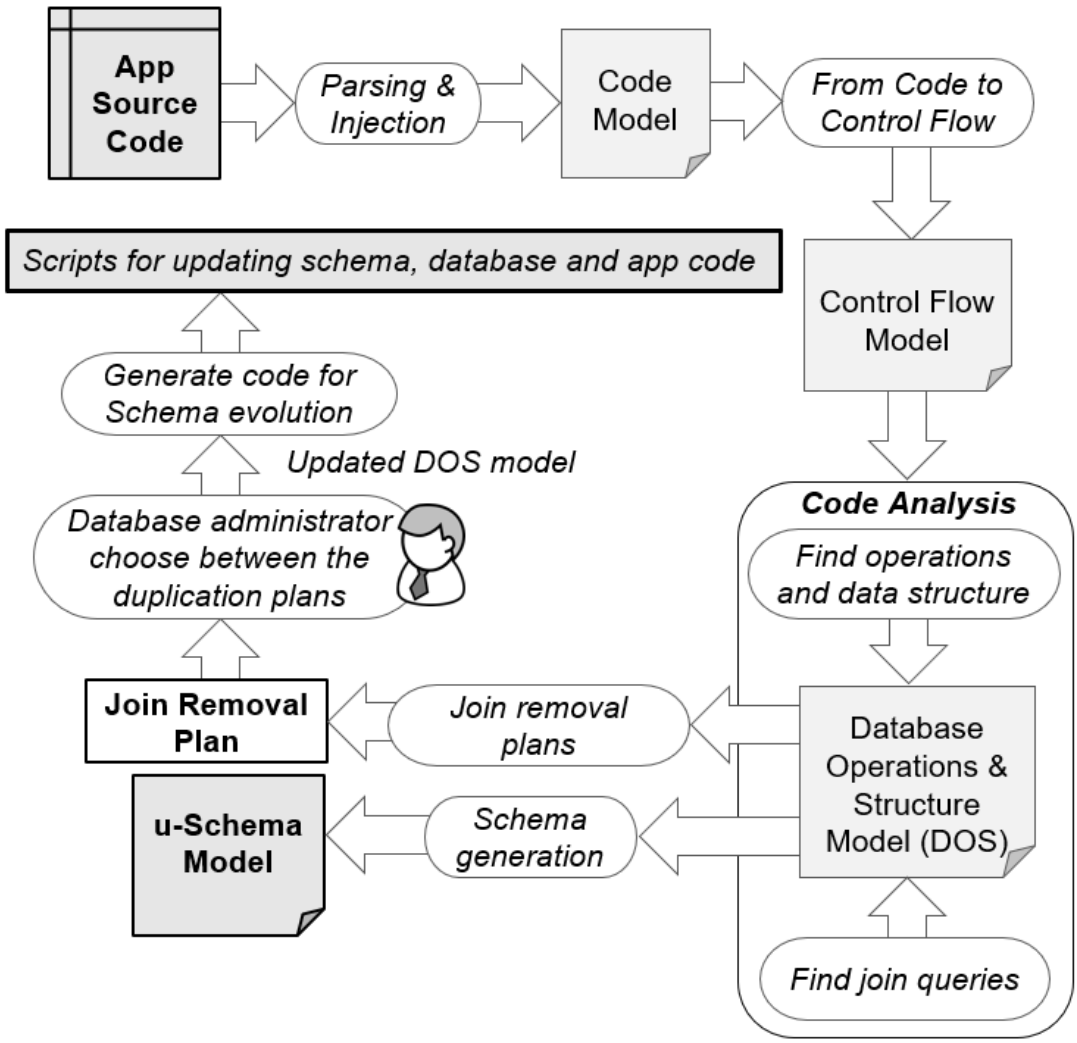

Figure 6.3: Overview of the U-Schema extraction and join query removal approach.

cessed document store. Code from lines 1 to 7 initializes the client variable that holds the connection from the client side to a MongoDB database. In line 9, starts the code that corresponds to the pseudo-code in Listing 6.1. It is expressed as a `findOne()` query issued on the `Users` collection, which has two arguments: the condition and a code block (i.e, a lambda expression) whose argument is the object returned by the query. That code block includes another `findOne()` query issued on the `Movies` collection. In this nested `findOne()`, the first argument establishes the join condition, and the second one is a code block containing an **if-then** statement whose code block is a sequence of three `console.log` statements.

Therefore, we have a nesting of three code blocks.

6.2 Obtaining an abstract representation of the code

In this section, we will explain how the source code to be analyzed is represented by means of the Code and Control Flow metamodels.

6.2.1 From source code to Code models

Instead of using the abstract syntax tree (AST) of the source code, we have defined the *Code* metamodel to represent the code in a more abstract form that is independent of the concrete syntax. An excerpt of this metamodel is shown in Figure 6.4, which includes the main concepts and relationships of the object-oriented languages, and covers the Javascript statements to be considered in our work. The decision of creating this metamodel is in accordance with the idea of building an abstract syntax's metamodel to create software languages [57, 61, 117]. In the definition of the Code metamodel, we had in mind the Java MODISCO [25] metamodel, and, to a lesser extent, by the *Code* package of KDM [68]. It should be noted that the metamodel was not designed for a particular object-oriented language but the specificities of the language are added as a separated metamodel.

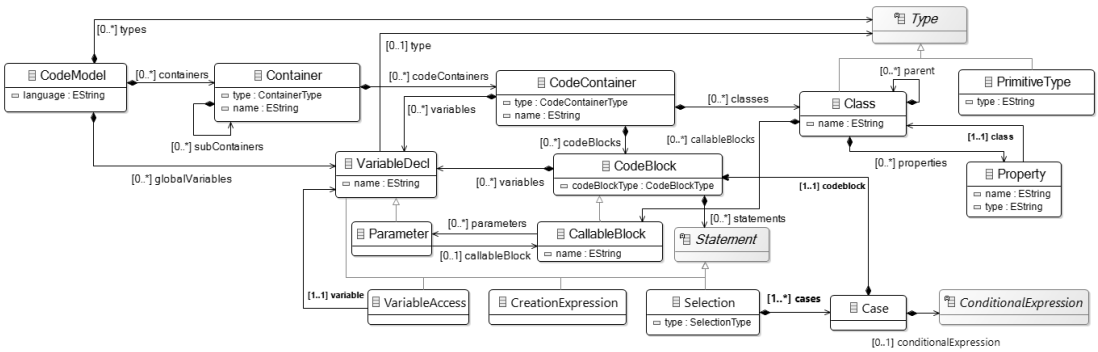

Figure 6.4: Excerpt of the main elements of the Code Metamodel.

Next, we will describe the Code metamodel to understand how Control Flow models are obtained. The code metamodel contains 55 metaclasses to represent JavaScript code, which includes the main concepts and relationships of the language abstract syntax. A *Code model* represents an executable piece of code as an Javascript script or program. It aggregates *Con-*

*tainer*s, global variable declarations (*VariableDecl*), a set of exceptions (*Exception*), and *Type*s that can be primitive types or classes, and a set of different types of code managing *containers* (*Container*): packages, nameSpaces, folders, and files. A container represents structures containing scripts or classes such as packages, namespaces, folders, and files. Containers can be nested and they aggregate *CodeContainer*s that, in turn, aggregate code blocks, classes, and variable declarations. A *CodeBlock* has a ordered list of *Statement*s, as conditional selection or loops, and also records local variable declarations. Figure 6.4 shows the main kinds of statements of code metamodel, the metaclasses related with variables in figure 6.5 and those related with statements in figure 6.6. Figure 6.4 also shows shows the statements that appear in the running example: conditional selection, method call, variable access and expressions to create objects. A special kind of code block is *CallableBlock*: code blocks that are invoked to be executed such as methods, functions, constructors and lambda expressions. There are four kinds of callable units (*CallableBlock*): functions, methods, lambda expressions, and constructors. A *CallableBlock* is a subclass of *CodeBlock*. Javascript code statements are represented as subclasses of *Statement* such as: *Loop*s, function *Call*, *Assignment*, conditional *Selection* with *IF* and *Switch* subclasses, and exception handling (*Try*, *Catch*, *Throw*).

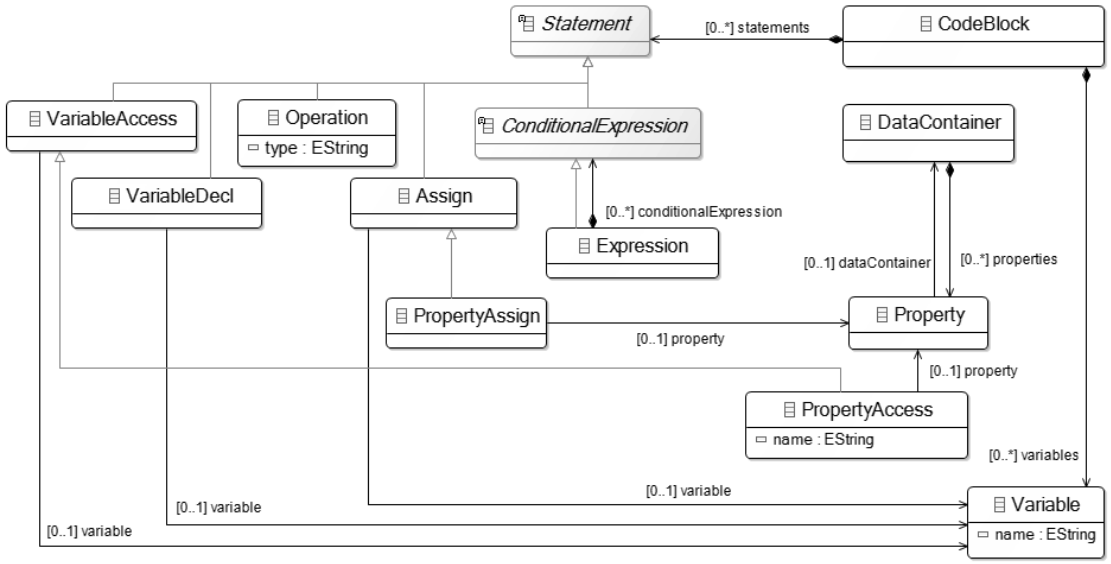

Figure 6.5: Excerpt of the variable related metaclasses of the Code Metamodel.

Figure 6.5 shows some of the metaclasses that manipulate variables, there is some variable

specializations like. A `Variable` will be local to a block if it belongs to a `CodeBlock`, `Global` if it belongs to a `Code`, `Parameter` (Figure 6.6) if it is the parameter of a `CallableBlock` or `Attribute` if it belongs to a `CodeContainer` class. There is also `Constant` for constants.

A `CodeBlock` has a set of `Statements`. In Figure 6.5 we can find `VariableAccess` to represent the access to a variable, `PropertyAccess` for the access to a property (`Property`) of an object (`DataContainer`), `ArrayAccess` (not shown in the diagram) for accessing an array position and `VariableDecl` for new variable declarations. For the assignments we can find `Assign` for the assignment of values represented as `DataProducer` (not shown in the diagram) to variables, `PropertyAssign` for the assignment of values to object properties. Another example of `DataProducer` is `Operation` from which different metaclasses inherit for mathematical expressions or string manipulation. For conditional expressions, one of the classes that inherit from `ConditionalExpression` is used, the diagram shows `Expression` that can contain a set of `ConditionalExpressions`.

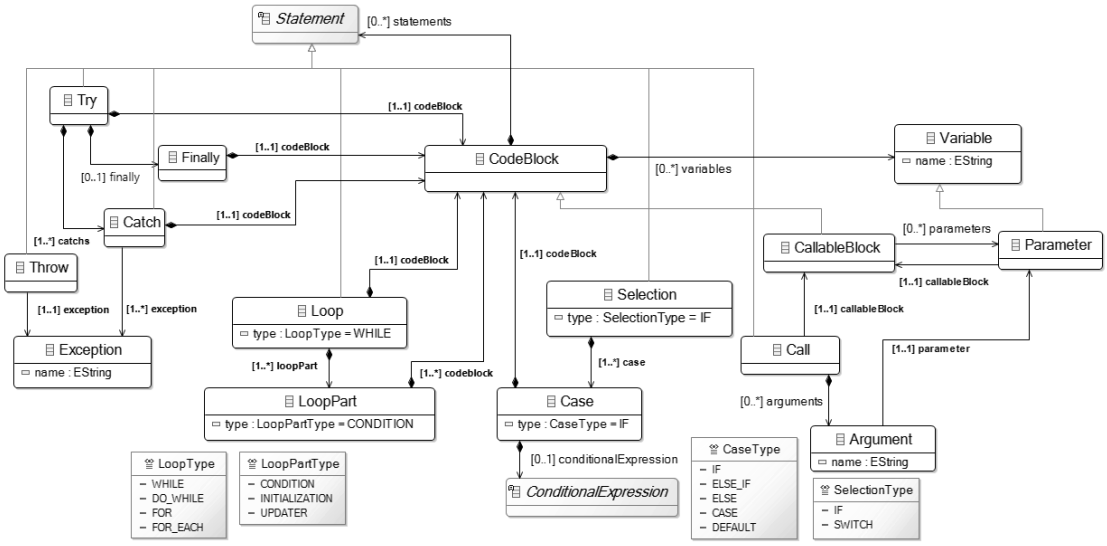

Figure 6.6: Excerpt of the Statements metaclasses of the Code Metamodel.

Figure 6.6 shows the main metaclasses of `Statements`. Among them we find `Try`, `Catch`, `Finally` and `Throw` for handling exceptions (`Exception`). `Loop` for the different types of loop (they are all represented with the same metaclass), `Loop` contains a set of `LoopPart` to indicate the different parts of a loop such as the condition, the updater or initialization. A similar case is `Selection` which is used to represent conditional statements as ifs or

switchs, a `Selection` has a set of `Case`s for indicate each of the possibilities and each one has an associated `ConditionalExpression`. To the right of the figure we find the representation of the elements that can be called (such as functions, arrow functions, anonymous functions, procedures or methods). A `CallableBlock` has a set of `Parameter`s to indicate the parameters and when a call is made with `Call` it is possible to specify the value of those parameters with `Argument`.

Code models are obtained (injected) from the AST provided by a parser. At the present, we have built an injector for Javascript, and the chosen parser was Esprima*. This parser provides the AST in form of a set of JSON documents. Therefore, we defined and implemented a mapping between the JSON object types of Esprima and the elements of the Code metamodel.

Figure 6.7 shows an excerpt of the Code model injected for the Javascript code of the running example. The figure does not show the part of the model starting from the *CallableBlock* element, which represents the lambda expression passed as second argument of the *findOne()* query. The injected model has a *CodeModel* root element that aggregates the container created for the file of the script "runningExample.js"; this container records the absolute path of the file. For this script, a *CodeContainer* is created whose type is "script". The code container only aggregates a code block that corresponds to the script itself, which represents the dot notation expression that includes the outer `findOne()` call: the *client* variable is accessed to invoke the *db* method call with *dbname* argument (another *VariableAccess* element), and that call is connected to the *collection* call that has the 'Users' literal as argument, and this call is finally connected to the 'findOne' call that has two arguments: an anonymous function *CallableBlock* for the lambda expression and an object creation expression formed by the "name" property and the "Brian" literal.

The complete Code model includes two *Class*es that represent the `user` and `movie` objects. Each class has a set of properties: `_id`, `title`, and `director` for `movie`; and `name`, `surname`, `email` and *watchedMovies* for `user`.

*Esprima website: https://www.esprima.org.

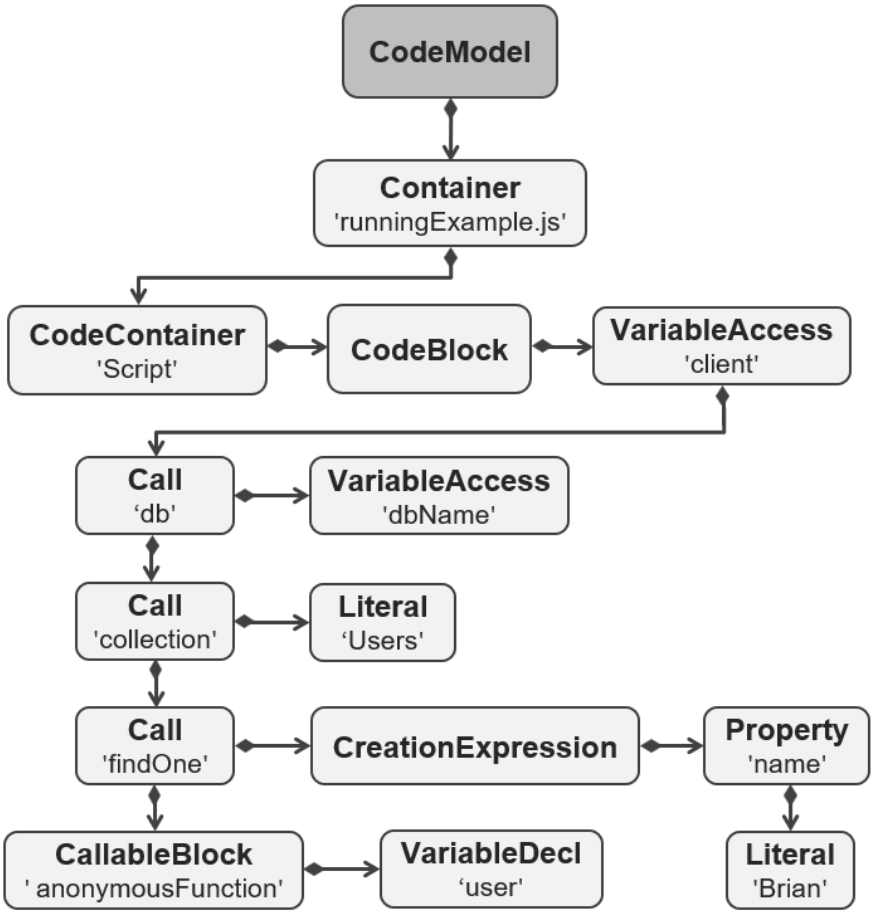

Figure 6.7: Excerpt of the *Code* model extracted for the running example (starting from line 9).

6.2.2 Testing

To validate this stage, we have applied the testing strategy defined in [28], where a software reengineering process is described. Some simple tests are performed for very small code snippets that only have the minimum instructions necessary to represent a single code statement, e.g., a loop. In each of these tests, the code is automatically re-built from the Code model generated, and the generated code is compared with the original code having into account the code format. In this way, we have carry out an iterative process, where the injection of a single statement was implemented and tested in each iteration. A Javascript code comparison tool was used to compare original and generated code.

Note that the iterative creation of the Code metamodel has prevented of using a model-

based language workbench to automatically generate the model injector from a EBNF-like Javascript grammar specification. This choice was motivated by the complexity and large size of the metamodel.

6.2.3 Representing the control flow

Commonly, a code analysis not only requires to work with a representation of the syntax tree, but also the knowledge of the control flow graph. To represent the control flow in our approach, we have defined the metamodel exposed in Figure 6.8. This metamodel has been designed having into account the representation proposed in the algorithm described in [11]. A *Control Flow* model is obtained from a Code model, and its nodes and edges will hold references to statements of the Code Model. Both models are input to the code analysis process described in the following section.

As seen in Figure 6.8, a Control Flow model contains a set of code subgraphs that can represent either code blocks or callable units (e.g., methods or functions). A code subgraph has nodes that corresponds to statements that are connected by edges: a node is source of outgoings edges, and target of incoming edges. Edges can be unit calls or conditional expressions. This means that nodes hold a reference to a Statement of the Code Model, and edges hold a reference to a call or conditional expression of the Code Model. Being connected a Control Flow model to the Code model from which was created, the code analysis can proceed by following the control flow to access to statements in the code.

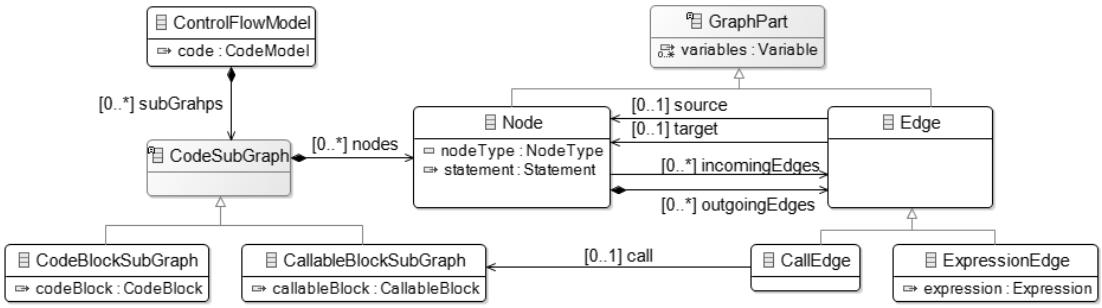

Figure 6.8: Control Flow Metamodel.

To obtaining Control Flow models, we have adapted the algorithm described in [11]. Here, the input is a Code model instead of an AST, and the output is a Control Flow

model. With these considerations, we have defined the Algorithm 1 to traverse an input Code model and create the nodes and edges of the output Control Flow model. The kinds of nodes and edges created will depend on the element processed in the Code model.

Algorithm 1 works as follows. Firstly, the *ControlFlowModel* root element is created (line 1), and all the code blocks (*CodeBlock*s and *CallableBlock*s) in the Code model are obtained and collected in a list that is iterated (lines 2 and 4). For each code block a *SubGraph* is created, which is initialized with its *start* and *end* nodes (lines 5–8). Statements of each code block are iterated (line 9), and a node is created for each statement by calling the `createConnectedNode` function (line 10) that, in turn, invokes the `createNode()` function (line 17). Each node references to the code statement and the variables used. By default, a newly instantiated node is connected to the last node created (line 11), but if the code statement is a conditional, a loop or a exception trigger, the `createConnectedNode` function creates new subgraphs to represent nested statements before creating new edges (lines 16–38). For example, a conditional is resolved as follows (lines 22–27): first, the initial node is created (line 23), then a new branch is created for each *Case* (line 25), and an conditional expression edge holds the condition for each branch (line 26). The `Branch` function instantiates the nodes that belongs to a branch, that is, nodes for the statements of the code block associated to the branch (lines 40–45), and the final node of the branch is also created (line 46). Finally, an edge is created to connect the last node created with the end node of the graph (line 47).

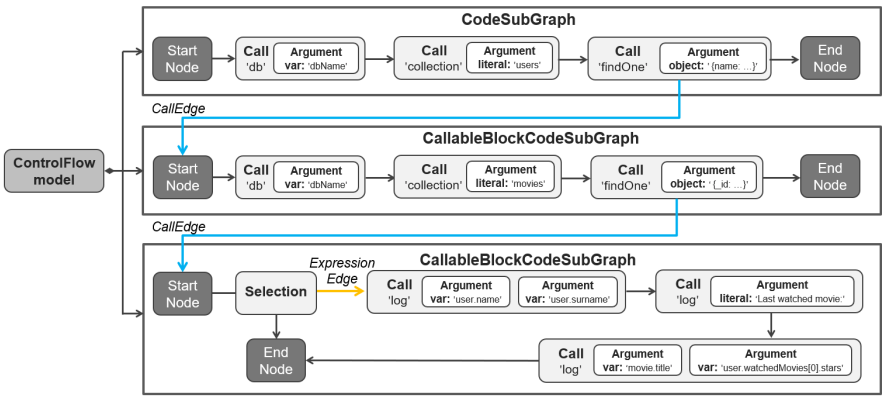

Figure 6.9: *Control Flow* model for the the running example.

The Control Flow model obtained for the Code model of the running example is shown in

```
   Data: codeModel : Code model
   Result: cfModel : Control Flow Model
 1 cfModel ← createControlFlowModel()
 2 codeBlocks ← getCodeBlocks(codeModel)
 3
 4 foreach cBlock ∈ codeBlocks do
 5 |   sGraph ← createSubGraph(controlFlow)
 6 |   sNode ← createStartNode(sGraph)
 7 |   lastNode ← sNode
 8 |   eNode ← createEndNode()
 9 |   foreach st ∈ cBlock.statements do
10 |   |   node ←
   |   |     createConnectedNode(st, st.variables)
11 |   |   createEdge(lastNode, node)
12 |   |   lastNode ← node
13 |   end
14 end
15
16 Function createConnectedNode(st, variables):
17 |   node ← createNode(st, variables)
18 |   if hasArguments(st) then
19 |   |   foreach arg ∈ st.arguments do
20 |   |   |   createEdge(node,
   |   |   |     createNode(arg, arg.variables))
21 |   |   end
22 |   else if isSelection(st) then
23 |   |   bNode ← createStartNode(node)
24 |   |   foreach case ∈ st.cases do
25 |   |   |   nNode ←
   |   |   |     branch(case.statements, bNode)
26 |   |   |   createEdge(bNode, nNode)
27 |   |   end
28 |   else if isTry(st) then
29 |   |   bNode ← createStartNode(node)
30 |   |   foreach catch ∈ st.catchs do
31 |   |   |   nNode ←
   |   |   |     branch(catch.statements, bNode)
32 |   |   |   createEdge(bNode, nNode)
33 |   |   end
34 |   else if isLoop(st) then
35 |   |   nNode ← branch(loop.statements, node)
36 |   |   createEdge(nNode, node)
37 |   end
38 |   return node
39
40 Function branch(st, lastNode):
41 |   foreach newST ∈ st.statements do
42 |   |   node ←
   |   |     createNode(newST, newST.variables)
43 |   |   createEdge(lastNode, node)
44 |   |   lastNode ← node
45 |   end
46 |   endNode ← createEndNode()
47 |   createEdge(lastNode, endNode)
48 |   return endNode
```

Algorithm 1: Control Flow creation algorithm

Figure 6.9. The *ControlFlowModel* root element aggregates three subgraphs: A *CodeBlockSubGraph* corresponds to the `findOne()` method call chain "client.db(dbName).collection('Users')

.findOne (name:name *nested lambda expression*)"; this subgraph therefore includes three "call" nodes in addition to the "start" and "end" nodes. The third "call" node has an outgoing edge whose target is the "start" node of a *CallableBlockSubGraph* that corresponds to another `findOne()` method call chain: client.db(dbName). collection ('movies'). findOne (_id: user . watchedMovies [0]. movie_id , *nested lambda expression*). Therefore this second subgraph also contains three "call" nodes, and the third one has an edge directed to the start node of the third *CallableBlockSubGraph* that includes a selection node with three call nodes that corresponds to the block of three println statements.

6.2.4 Testing

We have validated this second step by visually checking that obtained models accurately represent the control flow of the code. To achieve this, the models are stored as a Neo4j graph database in order to take advantage of Neo4j Browser,[†] which displays graph query results as graphs which can be navigated. Figure 6.10 shows an excerpt of the graph obtained for the Control Flow model of the running example. Each node contains the code snippet as a value. In particular, the graph represents the control flow for the *if-then* statement of the running example, which contains a block of three *console.log* statements. The `if` node has two outgoing edges whose types are: 'Selection' to record the condition, and 'Jump' that is directed to the node where the execution must continue if the condition is not satisfied. Each node labeled 'print..' has an outgoing edge labeled "NEXT" that establishes the flow of execution, and another edge labeled "argument" whose target is a node with the argument. These Neo4j graphs are more legible and easy to understand than the Control Flow model, and therefore they made the testing easier. The mapping to generate a Neo4j database from a Flow Code model is very direct, and the code is automatically obtained by applying the code generation used to validate the code model injection stage.

6.3 Discovering the database schema

Code and Control flow models are analyzed to discover the implicit database schema as well as to apply a database refactoring. In a first step of the code analysis, information about

[†]Neo4j Browser website: https://neo4j.com/developer/neo4j-browser.

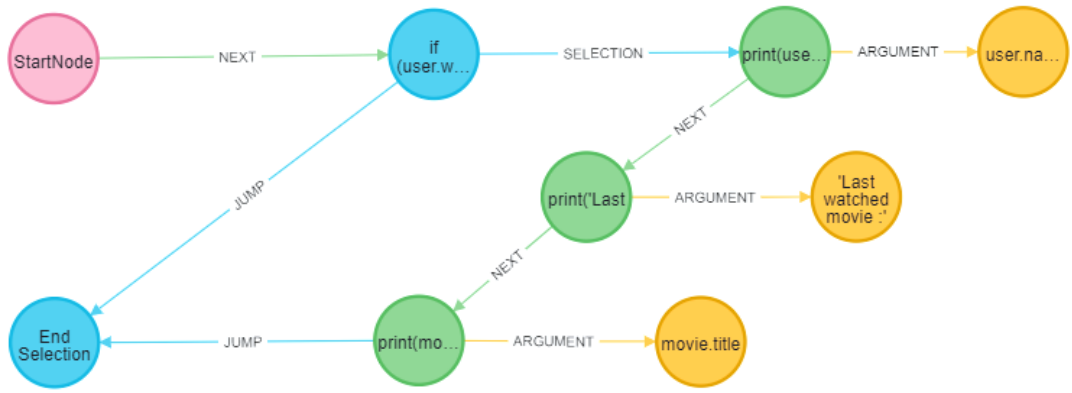

Figure 6.10: Graph excerpt of the Control Flow model for the running example.

CRUD operations and the structure of the data they manage is captured in an intermediate model. To serve this purpose, the Database Operation& Structure (DOS) metamodel has been defined, which is shown in Figure 6.11. In this section, we will first present the algorithm devised to obtain a DOS model, and then we will describe how a DOS model is transformed into a U-Schema model, i.e. a NoSQL logical schema. In the next section, we will show how the DOS model can be applied to apply the join query removal refactoring to improve the query efficiency.

6.3.1 Finding database operations and structure

DOSmodel is the root element of the DOS metamodel, as is observed in Figure 6.11. It aggregates *OperationDatabases* and *Containers*. A *Container* holds data structures, as collections in document stores or tables in relational databases. In turn, *DataStructures* aggregate the set of fields that have the objects stored in its container. A container can have more than one data structure because NoSQL systems can be schemaless, and therefore structural variation is possible. The approach here presented could be applied to different versions of the same application, and then structural variation could be identified, as is considered in [86]

Each *Field* has a name and a type. The type can be *Attribute*, *Collection*, *Aggregate* or *Reference*. An *Aggregate* type aggregates a *DataStructure* that represent the structure of embedded objects in a root object, and a *Reference* type holds a reference to the referenced attributes that belongs to another data structure. Regarding database operations, there is a subclass

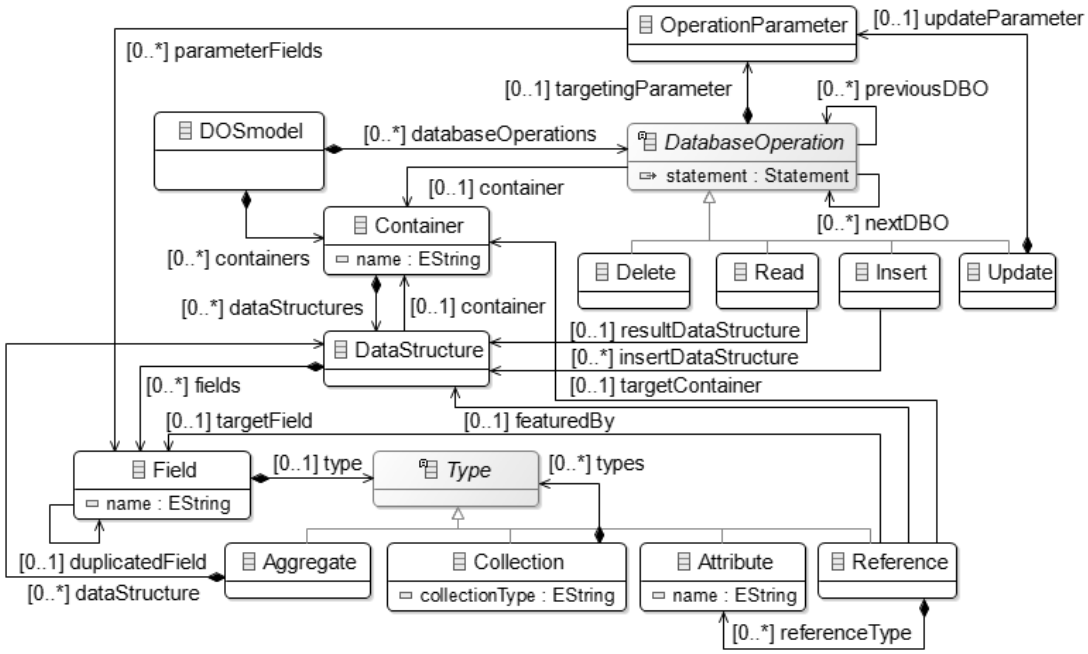

Figure 6.11: Database Operation&Structure Metamodel.

of *DatabaseOperation* for each CRUD operation: *Read*, *Insert*, *Update* and *Delete*. Note that *DatabaseOperation* holds a reference to a *Statement* of the Code model. *DatabaseOperations* also reference to the managed data structures, and can have parameters. Through *previousDatabaseOperation* and *nextDatabaseOperation* relationships, *DatabaseOperations* form a chain that follows the order in which they are executed.

A graph traversal algorithm is applied on the Control Flow model to find all references between the data and what data are involved in database operations. As the Algorithm 2 shows, a backward traversal is performed to create operations and dependencies between data, while a forward traversal discovers the data structures and connects operations and data. In Control Flow models, the nodes have outgoing edges which have a target node, and nodes also have incoming edges coming from source nodes, as shown in the Control Flow metamodel (see Figure 6.8). Source and target references of edges make it possible to perform the backward and forward traversals, respectively, as illustrated in Figure 6.12. In this figure, an blue edge leaves node A to node B through an *outgoing edge* and a *target* reference, and a red edge enters to node A from node B through an *incoming edge* and a *source* reference.

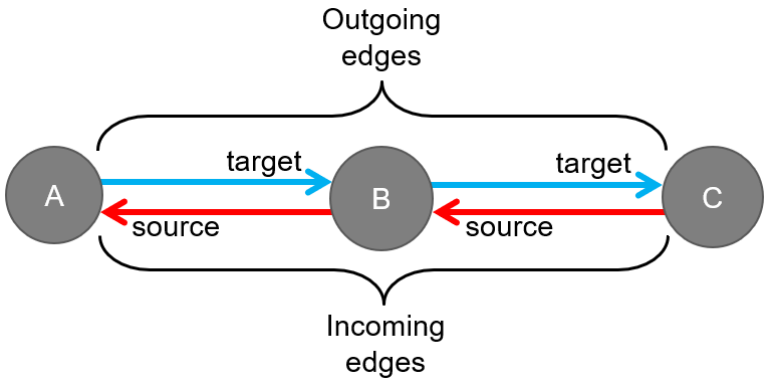

Figure 6.12: Nodes and edges in Control Flow models.

Before traversing the control flow graph, two operations are executed. First, a *DOSmodel* root element is created (line 1) Second, each subgraph (function o script) is traversed to obtain the *Call* nodes involved in each database operation. These nodes are collected in the "dbCallNodes" ordered list (line 2), where they are ordered by the execution order. These nodes are found by searching for the name of the function called, which should be part of the database management API used in the code. Note that, in a Control Flow model, each subgraph is connected to the following graph in the control flow through an *CallEdge*, as shown in Figure 6.9. Source and target subgraphs could belong to different functions or files.

After that initialization, the functions that implement the backward and forward traversals are called (lines 3 and 4). The `backwardTraverse()` function (lines 7 to 24) iterates the `dboNodes` list of *Call* nodes. For each visited node (`dboNode`), a *Read*, *Insert*, *Update* or *Delete* is first created depending on the kind of the database operation (line 9). Note that the instantiated *DatabaseOperation* (`dbo`) will contain a reference to the statement of the Code model, this statement is obtained from the node. Next, *Arguments* of `dboNode` are recorded in a `sList` *search list* (line 16) and the graph is traversed backwards from the source node to the current node (lines 12–13). Each node `sNode` visited in that traversal is processed as follows. If it corresponds to a database operation call and the variable storing the returned value by that operation call matches to one of the variables in the search list `slist` (line 14), then the previously instantiated operation (`dbo`) is connected to the operation created for *sNode* (*pDBO*) through the *previousDBO* relationship (line 16), and this second operation will

refer to `dbo` through the *nextDBO* relationship (line 17). When two operations are connected in that way, this means that they are dependent due to that the data outputted by a operation are input to the other one. If both operations are issued on different collections, then the operation receiving data is marked as a join query (line 18). Is convenient to note that a function *findDbOperation* is in charge of obtaining the statement referenced by the visited node (*sNode*). This function navigates to the Code model to return the database operation call statement (*Call*); Next, the set of database operations of the DOS model is iterated to check if the call operation is present. When the visited node `sNode` do not satisfy the condition indicated above, if the *Statement* associated to the node is an assignment and if the variable on the left side of the assignment is present in the list `sList` (line 19), the variable on the right side of the assignment is added to the `sList` list (line 20).

Once the backward traversal is completed, all the database operation *Call* nodes are again visited through a forward traversal (line 4). In this traversal are only processed nodes of *Read* operations by means of the function `forwardTraverse()`. For each Read node (line 28), a *DataStructure* is instantiated and aggregated to a *Container*, both instances are created if they do not exist (lines 29 and 30). Also, the *DataStructure* is linked to the Read operation (line 31). A `search list` is created in this traversal, but it is intended to contain the data retrieved from Read operations (line 32).

Each subgraph is traversed (lines 34 to 42) starting from the next node of the database node (`tNode`) (lines 34 and 35). For each visited node (`tNode`), its list of variables is iterated to check what variables are holding values read from the database. For this, each variable is matched against items of `sList` (line 36). When a match occurs, the Read *Statement* is analyzed to find the fields of the object retrieved. When a property access is found, a new *Field* is created and associated with the current *DataStructure* (lines 37–39). The type of the *Field* would established as follows (line 38):

- if another property is accessed from the current property, this denotes that the current property holds an embedded object and its type is *Aggregate*.
- if a collection operation is found, the type of the current property is *Collection*.
- otherwise the type of the current property is *Attribute* (References are not detected during the forward traversal of the graph).

Note that the fields of a particular data structure can be found at any point of the traversal, and even the type of a field could change or not be found

Once the forward traversal is completed, it is carried out the search for attributes that are really references by calling the function `createReferences` (line 5). This function first collects the `Read` operation marked as "join query" (line 46), i.e., those whose *previousDatabaseOperation* relationship is not null. This collection is iterated (lines 47-51), and for each join query, the join condition is inspected to extract the name of the field involved in the join (lines 48 and 49). Thus, a *Reference* type in the DOS model is created (line 50). This Reference type references a target container, and it is also associated to the Attribute field identified in the forward traversal. Note that references cannot be extracted at the point that join queries are identified because its data structure has not been created yet. Finally, it is checked if several data structures have the same set of fields, only one of them is kept and the rest are discarded.

Figure 6.13 shows the DOS model obtained when applying the Algorithm 2 to the running example. A *DOSmodel* aggregates two *Read* operations, one for each `findOne` call node, and the containers *movies* and *users*. In turn, each *Read* operation aggregates its data parameter, and each container its data structure which has the fields of the objects stored in the corresponding container.

This DOS model would be created as follows. In the backward traversal, two *Read* operations would be discovered and they reference each other through *next* and *previous* relationships. These two operations would comply the condition to detect a join query, which would marked. After, in the forward traversal, two *Container*s are found and each has its own *DataStructure*. These containers would be identified by inspecting the arguments of the READ operations. Regarding the detection of fields, (i) the `name` field of User and the `_id` field of *Movie* would be found, respectively, in the first argument of the first and second *Read* operation, and (ii) the rest of fields would be found when the result variable of each query is used in the *if-then* Selection node: condition expression of and its only branch, and statements `console.log` of three Call nodes that form the block associated to the branch (see Figure 6.9) For the `watchedMovies` field would be detected an array collection that contains elements of a third data structure. The type of the `watchedMovies` field would be an array of `Aggregate`s of objects of a third data structure with the fields `stars` and

```
Data: cfModel : Control Flow Model
Result: dosModel : DOS Model
1  dosModel ← createDOSmodel()
2  dboNodes ← getDatabaseOpsCallNodes(cfModel)
3  backwardTraverse(dboNodes)
4  forwardTraverse(dboNodes)
5  createReferences(dboNodes)
6
7  Function backwardTraverse(dboNodes):
8      foreach dboNode ∈ dboNodes do
9          dbo ← createDatabaseOperation(dboNode)
10         sList ← getArguments(dboNode)
11
12         sNode ← getPreviousNode(dboNode)
13         while ∃ sNode do
14             if isDatabaseOperation(sNode) ∧ getReturnVariable(sNode) ∈ sList then
15                 pDBO ← findDbOperation(sNode)
16                 dbo.previousDBO ← pDBO
17                 pDBO.nextDBO ← dbo
18                 markJoinQuery(pDBO)
19             else if isAssignment(sNode) ∧ getLeftVar(sNode) ∈ sList then
20                 sList.add(getRightVar(sNode))
21             end
22             sNode ← getPreviousNode(sNode)
23         end
24     end
25
26 Function forwardTraverse(dboNodes):
27     readNodes ← getReadsOperations(dboNodes)
28     foreach dboNode ∈ readNodes do
29         container ← getOrCreateContainer()
30         ds ← getOrCreateDataStructure(container)
31         dboNode.result ← ds
32         sList ← dboNode.statement.result
33
34         tNode ← getNextNode(dboNode)
35         while ∃ tNode do
36             if tNode.variables ∈ sList then
37                 field ← createField(tNode)
38                 field.type ← createType(tNode)
39                 ds.fields.add(field)
40             end
41             tNode ← getNextNode(tNode)
42         end
43     end
44
45 Function createReferences(dboNodes):
46     joinQueries ← getJoinQueries(dboNodes)
47     foreach dbo ∈ joinQueries do
48         sField ← findSource(dbo.previousDBO)
49         tField ← findTarget(dbo)
50         createReference(sField, tField)
51     end
```

Algorithm 2: Code analysis algorithm

`movie_id`. This second field would finally identified as a reference.

6.3.2 Testing

We tested the Algorithm 2 by using small pieces of code that were similar to the running example. For each piece of code, we manually checked that the *Container*s, *DataStructure*s, *Field*s, and *Type* were correctly represented in the DOS model.

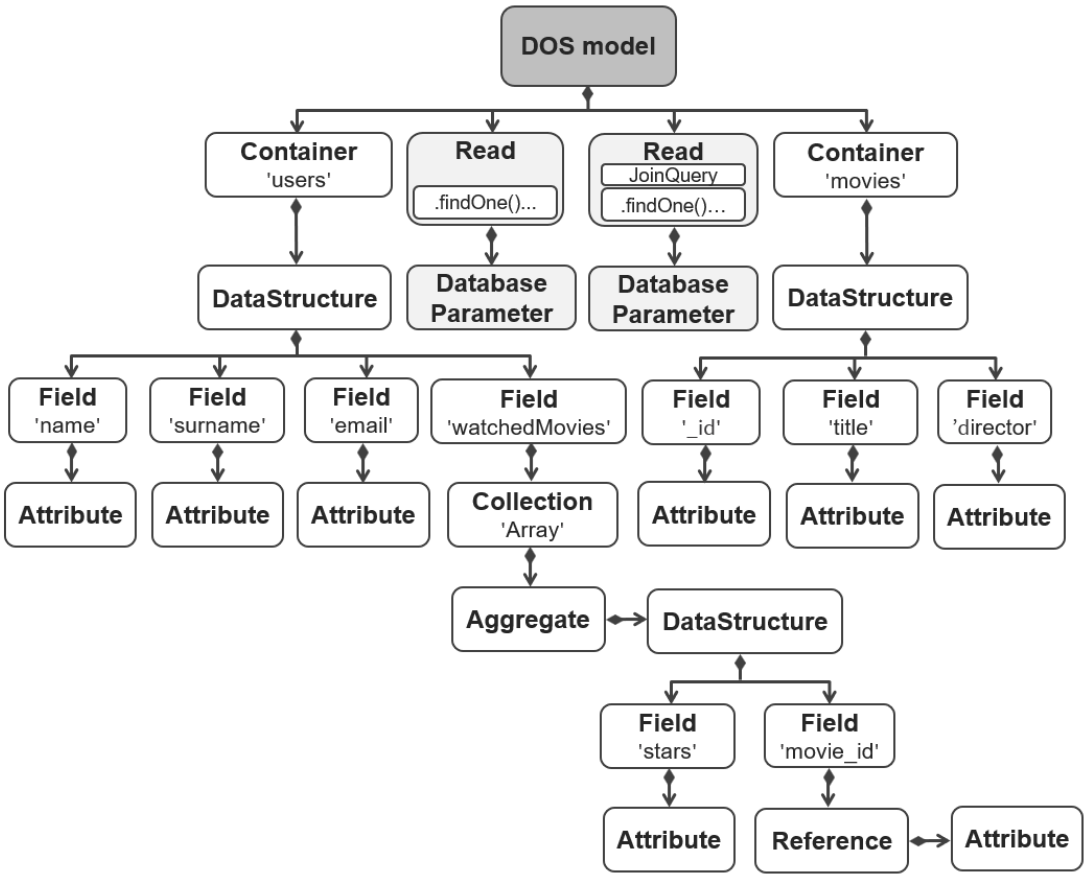

Figure 6.13: Database Operations & Structure Model for the running example.

We started creating a small code with one read database operation that only uses one property in the filter. We checked the model that contains a *Read* and a *Container* with a *DataStructure* containing only one *Field*. Next, we added statements manipulating the result object obtained from the query. This introduced new fields to be discovered, we review the model again to check if the added fields were represented. The process continued several steps by adding to the code new kinds of database operations and accessing new containers (at maximum of 3). In each new step, new fields were introduced to vary the data structures.

6.3.3 Generating schemas as U-Schema models

As explained above, the DOS metamodel represents both the set of database operations included in the piece of code analyzed and the structure of the data stored. Therefore, a

DOS Model	U-Schema
Container	Entity Type
DataStructure	Structural Variation of an entity type
Attribute field	Attribute feature
Reference field	Reference feature
Aggregate field	(aggregated) Entity Type, Structural Variation, and a Aggregate feature

Table 6.1: DOS metamodel to U-Schema metamodel Mappings.

DOS model captures the database physical schema: containers of objects whose data structure is formed by a set of fields that can be attributes, collections, aggregates, or references. With this structural information, a logical schema could be abstracted. We have used the U-Schema unified metamodel explained in Chapter 4, shown in Figure 4.1, to represent such schemas, and have implemented a model-to-model transformation to extract a logical schema from a DOS model.

Table 6.1 shows the DOS-to-U-Schema mappings that we applied to obtain logical schemas. The transformation first creates a U-Schema model as root element. Then, an *EntityType* is created for each *Container* element. After that, every *DataStructure* is mapped to a *StructuralVariation* of the U-Schema metamodel that will hold a set of features. The entity type could have more than one variation if the code analysis is repeated for different versions of the same script or application. *Field*s are mapped to U-Schema elements as follows: (i) an *Attribute* feature for each *Attribute* field, with the same name and primitive type. (ii) a *Reference* feature for each *Reference* field, which be connected to the *EntityType* instanciated for the *Container* referenced by the *targetContainer* reference. (iii) Each *Composition* field is mapped to an embedded (non-root) entity type, and an *Aggregate* feature. This also instanciate a *StructuralVariation* for the *DataStructure* referenced by the *Composition* field trough *dataStructure* reference. The process is repeteated for the all the fields in this *DataStructure*. (iv) An *Attribute* feature for each *Collection* field, the kind of the collection is obtained from the property *collectionType* of the *Collection* meta-class. Note that a collection could contain values or embedded objects.

Figure 6.14 shows the U-Schema model obtained for the DOS model in Figure 6.13. The schema is visualized with the NoSQL schema notation presented in Chapter 7. The schema

includes the *User* and *Movie* entity type (yellow boxes) that correspond to the `user` and `movie` *Containers*. Each entity type aggregates (black dashed arrows) only one *Structural-Variation* (white boxes) that comes from the *DataStructure* associated to each Container in the DOS model. A third entity type `WatchedMovie` (grey box) corresponds to the data structure embedded in the one with the `user` container. `WatchedMovie` also has only one structural variation, which references (blue solid arrow) to `Movie`. An aggregate relationship (red solid arrow with diamond) connects the `User` to `WatchedMovie`. Attribute features are shown inside the boxes representing structural variations.

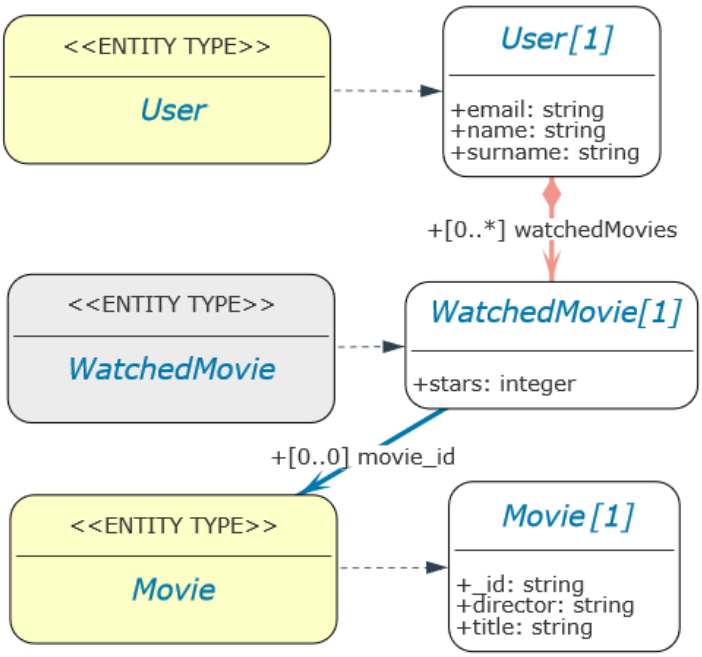

Figure 6.14: U-Schema model resulting of the code analysis.

6.4 Generating join removal plans

A DOS model provides information useful to help administrators to detect that some refactorings should be applied to improve data quality or query performance. For instance, knowing the number of fields of each entity type can help to detect what entity types are bloated and they should be divided, and join queries are useful to identify which separated data should form an aggregate to avoid a costly join between two containers.

To illustrate how our approach makes it possible the detection and application of refactorings, we have chosen the "join query removal" refactoring, which was described in detail in Section 6.1. Applying this refactoring requires the following information is available: join queries, the two involved entity types, and data to be duplicated. The latter is the only one not present in the DOS model, and, and we will here explain how discover it. We refer to the information provided to administrators for this refactoring as "join removal plan".

Algorithm 2 discovers and marks those *Read* operations that are join queries (line 12), as commented in Section 6.3.1. Therefore, the main task of the algorithm that creates join removal plans is to detect which fields should be duplicated for each join query in the DOS model. These fields will be found in the statements appearing after the join query in the control flow. They will be variable access statements in which the result variable of the join query is accessed (dot notation) to obtain the value of a particular field of the data structure associated to the container on which the join query is being issued. Note that the result variables of the two queries involved into a join are used in the same *CodeBlock*.

Algorithm 3 duplicates data that are necessary to remove a join query. It works as follows. First, the set of join queries (*Read*s) are obtained, i.e., those which refers at least one *prevDatabaseOp* relationship (line 1). Next, two variable search lists are defined (lines 2 and 3). For each join query, the result variable of the join query is added to the `joinSList` search variable list (line 7), and the variable result of the previous query is added to the other search variable list `prevSList` (line 8).

At this point, a function *findFollowingNode* obtain the node from the the control flow that correspond to the current join query, and return the following node (line 10). Then, a forward traversal is performed to find which variables are used in the statements that follow in the flow of control (lines 11 to 23). Each visited node is inspected to check whether any of its variables is included in both `joinSList` list and `prevSList` list (line 12). This condition is satisfied for a variable whenever that the join query's result is used together with the result of the other query, which means that the fields accessed from the result variable of the join query should be duplicated if that query is removed. Whenever that the condition is satisfied, the DOS model is updated as follows. The fields for the join query database operation used in the current statement are obtained from the visited node in the ControlFlow model (line 13), and the corresponding *Field* object is copied (line 14) and

assigned to the result of the previous *Read* database operation (line 15), i.e., the *DataStructure* of the object obtained in the *Read* database operation. The copied *Field* also has a reference to the original *Field*. While traversing the control flow graph, if an assignment of the result of the database operation to another variable is found (line 17), the new variable is included in the `joinSList` (line 18). The same happens for the previous database operation in lines 20 and 21. This process is repeated for all database operations.

When the algorithm is applied on the DOS model of the running example, the iteration of the join query list is limited to visit the single "join query" read operation in the model as seen in Figure 6.13. That operation is processed as follows. The (*movie*) result variable of the join query is added to the *joinSList*, and the previous query is accessed and its (*user*) result variable is added to *prevSList*. Next, the control flow graph is traversed starting from the node obtained by the function *findFollowingNode*. This function returns the node that corresponds to the *if-then* statement, which is labeled *Selection* in the Figure 6.9. After that, the following nodes are traversed to find the usage of each variable present in the variable search lists. That is, it is checked if the *user* variable in *prevSList* and the *movie* variable in joinSList) are both used together in a *statement* or *CodeBlock*. The algorithm will traverse the three nodes that represent the three *call* functions in the control flow model of Figure 6.9, and it will be found that both variables are used together in the last *Call*, and then the *title Field* is identified as a candidate to be duplicated because a *movie.title* expression is found. Thus, *title* is copied and added to the *DataStructure* to which belongs the *movie_id* variable involved in the matching of the join condition. This newly create field is marked as duplicated with a reference to the original *Field* in Movies.

Once data to be duplicated are discovered and the DOS model is modified, this model captures the information that requires an administrator to decide whether or not a particular join query is removed. A small application could collect the information of each join removal plan in order to be visualized on the screen, so that administrators can decide which of them are applied. For each plan, the following information is provided: join query and related queries, source and target entity of the join, data of the target entity which should be duplicated in the source entity, and original code along with a possible rewriting of the involved code. Whenever a plan is selected to be applied, a schema change operation must be performed, and this implies that schema, database and code must be

```
   Data: cfModel : Control Flow Model
   Data: dosModel : DOS Model
   Result: dosSModel : DOS Model
 1 joins ← getJoinQueries(dosModel)
 2 joinSList ← ∅
 3 prevSList ← ∅
 4
 5 foreach join ∈ joins do
 6 |   pDBO ← dbo.prevDBO
 7 |   joinSList.add(getResultVariable(join))
 8 |   prevSList.add(getResultVariable(pDBO))
 9 |
10 |   node ← findFollowingNode(cfModel, join)
11 |   while ∃ node do
12 |   |   if node.variables ∈ joinSList ∧
   |   |     node.variables ∈ prevSList then
13 |   |   |   fields ← getFields(dbo, node.variables)
14 |   |   |   joinQueryFields ← copyFields(fields)
15 |   |   |   pDBO.resultDS.add(joinQueryFields)
16 |   |   end
17 |   |   if isAssignment(node, joinSList) then
18 |   |   |   joinSList.add(node.variables)
19 |   |   end
20 |   |   if isAssignment(node, prevSList) then
21 |   |   |   prevSList.add(node.variables)
22 |   |   end
23 |   |   node ← getFollowingNode(node)
24 |   end
25 end
```

Algorithm 3: Finder of fields to be duplicated

updated.

6.4.1 Updating the database

Collections that receive duplicated data must be updated. This is achieved by using a template language to create a model-to-text transformation that specifies how the updating code is generated from the DOS model. The template iterates over the set of join queries, and for each one of them obtains the fields to be duplicated, and expresses the query to obtain this data and the operation to update the collection. This query is built with the *Field*s

duplicated in the DOS model. In these *Field*s, the reference *duplicatedField* points to the original *Field*. In this way, it is possible know the collections involved and the names of the properties to be duplicated. Listing 6.4 show the code generated for the data duplication of the running example.

```
const client = new MongoClient(url);
client.connect(conErr => {
  const db = client.db(dbName);
  db.collection("Movie").find().forEach(movie => {
      db.collection("User").updateMany(
        {watchedMovies: {$elemMatch: {movie_id: movie._id}}},
        {$set: {"watchedMovies.$[it].movie_title": movie.title}},
        {arrayFilters: [ {"it.movie_id": movie._id} ] }
      );
  });
});
```

Listing 6.4: Database update code generated for the running example.

6.4.2 Updating code

Code is updated in a 2-steps process. First, a model-to-model transformation is defined to update the Code model, whose input are the DOS and Code models and the output the modified Code model. Again, the DOS model provides the data to be duplicated for each join query, as well as the read operations that can be eliminated. These queries are removed, and then a replacement of code is performed in the expressions in which the result variable of the removed join query is used: the code of the *CodeBlock* contained in the second query is added to the previous one. Then, the variable is replaced by the result variable of the previous query followed by the field added in the duplication, but if this field is part of an aggregate, then will be necessary to access the duplicated data's field through the field of type *Aggregate*, as explained above for our running example. Note that the deletion and replacement can be done on the Code model as this model is referenced from database operations in the DOS model.

In a second step, the Code model is traversed to generate the updated code. This traversal had already been implemented as part of the testing of the Code model injection process in Section 6.2.

In the running example, the second query is a join query as the DOS model in Figure 6.13 expresses, and this query uses the `user` result variable of the previous query, as the Code

model in Figure 6.2 records. If this query is deleted then the `movie.title` expression would be replaced by the "user.watchedMovies[0].movie_title", according to the explanation above.

6.4.3 Testing

We tested the last algorithm by using the pieces of code created for testing the DOS model generation algorithm (Algorithm 2), but applying small changes. We have included join queries after some query already present in the code. Once the algorithm was executed, we inspected the modified DOS model to check that the *Read* database operations involved in the join queries have the references *previousDBO* and *nextDBO* are correctly set between them. Also, we checked that the *Field*s to be duplicated were correctly copied and referenced. As join queries were added, its complexity increased, and finally we added queries using the aggregation mechanism of the MongoDB API.

6.5 Validation

A testing strategy has been applied for each step of the reverse engineering process described in the three previous sections. In this section, we will show the validation carried out of the complete code analysis process, whose input is code that manipulates a NoSQL store, and the output is the database schema and a list of of join removal plans. We have also considered the application of schema change operations for join removals selected by database administrators. First, we will describe the experiment designed and the methodology followed for the evaluation, then we will present the results obtained, and we finally expose the limitations of our validation.

6.5.1 Experiment: Description and Methodology

The validation consisted in a round-trip experiment. We manually developed database access code of the backend of a small part of the functionality related to a music streaming service, namely rating and retrieving information about artists, albums and tracks. Data were stored in collections of a MongoDB database.

We defined the schema shown in Figure6.15 in form of a U-Schema model, and create the collections *artists*, *albums*, *tracks* and *genres* to store database objects. The collections were populated by using the automatic data generator presented in [75].

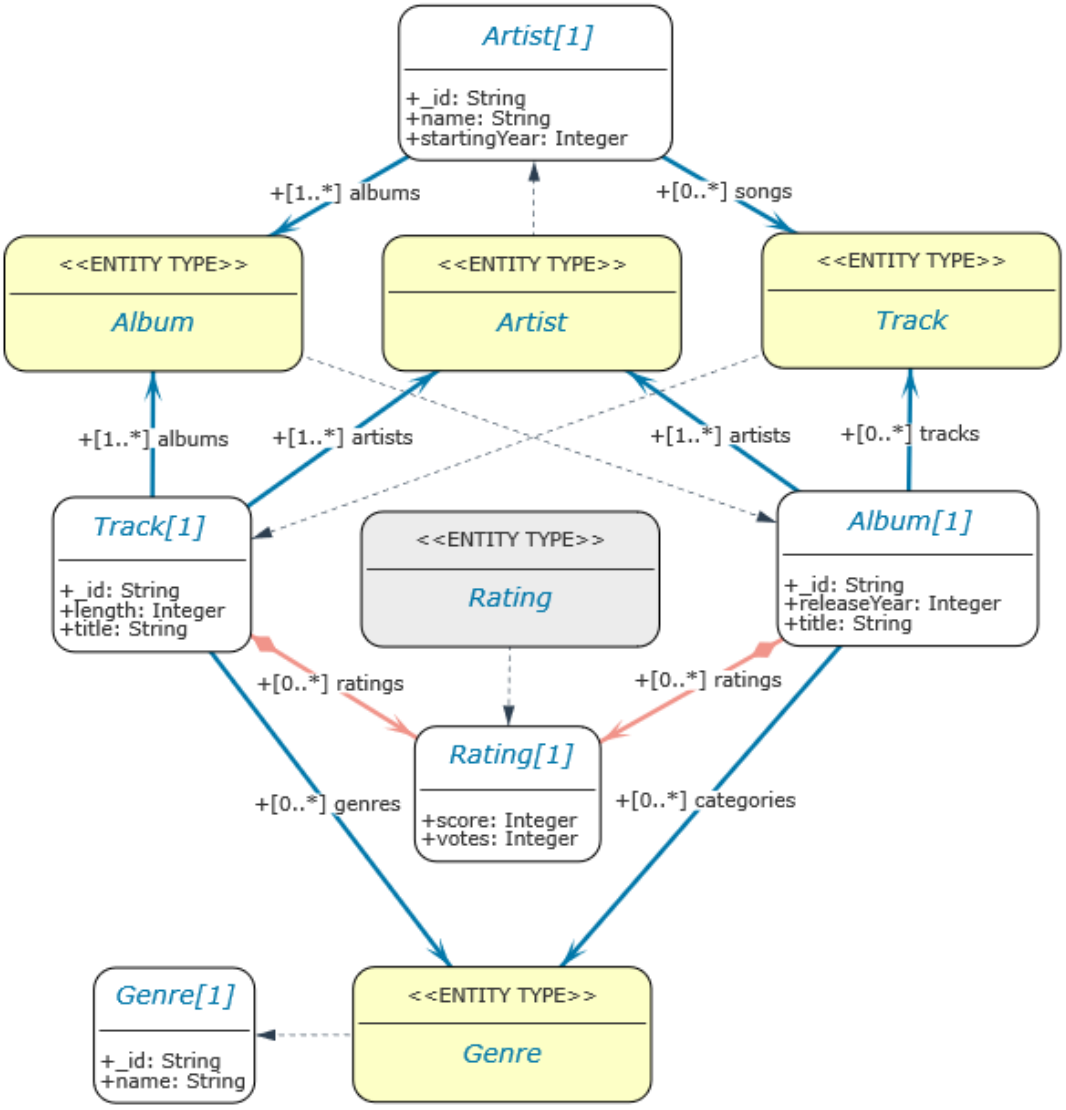

Figure 6.15: Validation designed schema.

The designed schema contains 5 entity types: *Album*, *Track*, *Artist*, *Rating* and*Genre*. *Artist* references to zero or more *Albums* and *Tracks*, *Album* references one or more *Tracks*, and both *Album* and *Track* references to one or more *Genres*. *Album* and *Track* aggregates zero or one *Rating*. Thus, *Rating* is an embedded entity, while the rest are root entities. The attributes of each entity type can be seen in Figure 6.15.

We then executed our code analysis solution taking the code developed as input. Firstly,

we compared the inferred U-Schema model with the schema previously designed by us and the extracted schema from the data analysis approach, defined in Chapter 5 for MongoDB. With these two model comparisons, we could check the correctness of the solution here presented, and to what extent it is more or less accurate that our data analysis approach.

Regarding the join removal refactoring, we first checked that all the join queries that we had written in the application were correctly detected. Next, we selected and applied each join removal, and we checked that schema, database and code were correctly updated for each removal, i.e. data were properly duplicated. Schema checking was performed in the following way. We changed the original schema by manually adding some duplication data, i.e. for each entity type t_1 referencing another one t_2, we added to t_1 the fields of t_2. Then, we compared this modified original schema with the one that resulted of applying the refactoring. In our case, the property `name` of the `Artist` entity type was copied into the `Album` and `Track` entity types, and the property `releaseYear` of `Album` was copied into `Track`.

The backend developed has 15 methods which support the functionality indicated above. Each of them has at least one database operation. The total number of database operations is 27: there is one insert, update and delete for each root entity type, and a total of 11 queries. Three of the query methods have a nested query in which the result the outer query is used in a join condition, and two of the queries use the *Aggregate* operation of the MongoDB API to perform a join between two collections as shows the Listing 6.5. In the insert and update operations, input data are checked to assure that they are properly formed according its corresponding field (e.g., if a quantity field is greater than zero, if a name field is not empty, or if a particular field is present.)

```
1 db.collection("album").aggregate([{
2   $lookup: {
3     from: "genre",
4     localField: "categories",
5     foreignField: "_id",
6     as: "categories"
7   }
8 }])
```

Listing 6.5: Join Query between Album and Genre expressed with an Aggregation operation.

The database access of the backend was implemented by using the official MongoDB

API,[‡] and the process was applied for each database access file. It was created a class for each entity type in the schema, and a Repository for each root entity type.

6.5.2 Results

Our solution has been able to correctly detect all database operations, including the aggregate operations. Also, the database schema has correctly been extracted. The schema is the same as in the predefined schema, however it is not the same schema that the inferred from the data analysis (Chapter 5).

If data with different structural variations had been generated, rather than data with the same structure for each entity in the schema, the code analysis could not have discovered this variability, which can only be detected if several code versions are analyzed. Instead, references if are detected from the code. In contrast, NoSQL data analysis approaches can discover structural variability, but the heuristics-based search for references can not guarantee that all they will be found and avoid some inconsistencies. In our case, the `songs` name does not match any of the applied heuristics. Instead, it was detected in the data analysis.

Analyzing the code of different versions of an application, the evolution of the schema can be traced as described in [86]. Achieving this with a data analysis is a challenging problem although the timestamp of data insert and update is available.

The code contained in the anonymous code blocks of the 3 join queries were correctly identified, and 3 join removal plans were created for them. These plans exactly corresponded to the join queries the variables to be duplicated were correctly discovered. When all the fields of an entity type whose objects are retrieved in a join query had to be duplicated, we created an embedded object in the object where those fields had to be duplicated in order to favor the data cohesion.

6.5.3 Limitations of the validation

Some potential limitations were previously determined. The schema is not complicated, but it was designed having in mind including all the elements that are part of U-Schema, and therefore of logical modeling, and more common modeling techniques or usages ap-

[‡]MongoDB API: `https://docs.mongodb.com/drivers/node`.

plied in real schemas. However, some techniques have been left out of the study as self-references, and other have been applied with some restrictions, as nesting query at two levels.

Note that a greater number of entity types and database operations would not necessarily imply a more reliable validation, as this would only suppose to repeat more times the process described in this paper, and affect to the number of collections (i.e., entity types) and fields.

The number of schema changes to duplicate data when removing join queries is low. However, this change is enough to check if the algorithm is capable of detecting possible properties to be duplicated.

7

A Generic Schema Query Language: SkiQL

When developing database applications, developers have to access database schemas to obtain information about the data structure. Therefore utilities for querying and visualizing schemas are essential. In this chapter, we will describe the SkiQL query language and the graphical notation we have designed and implemented with the purpose of providing such an utility for schemas represented as U-Schema models. Before explaining the syntax and semantics of SkiQL, we will define some kinds of schemas and show the graphical notation devised to show the query results (i.e., sub-schemas). Once SkiQL is described in detail, we will explain its implementation, and we will end the chapter showing the evaluation carried out.

7.1 Kinds of NoSQL Schemas

Most NoSQL schema inference approaches [119, 84] do not extract variations or relationships between entities. Thus, the schema notion considered is the set of *union entity types*, where each type is formed by the union of all the features that are present in its variations. Instead, a U-Schema schema contains a set of schema types, with their variations, and the relationships (aggregation and references) between variations or types. Next, we define some sub-schemas and schema views that are of interest in providing useful information

on the stored data structure.

Type-Variations Sub-schema It contains a schema type and all its variations. All the properties of a variation are considered pairs formed by its name and type, and the cardinality is also included for aggregations and references.

Union Type It is a reduced view of a type-variations sub-schema that results of gathering all the variations of the type into a single variation. The set of features of such a variation is the union of the set of features of the gathered variations. Obtaining the union of features requires making a decision to resolve feature name collisions: the same feature name is associated to different types in different variations of the same entity type. We decided to use union data types, i.e., a feature can have multiple types.

Simple Schema of Union Types Joins all the union types of a schema. As mentioned above, this kind of schema is the result commonly obtained in the extraction approaches for document stores, such as [84] and [119]. Actually, it is not a schema, but a reduced view of a NoSQL complete schema.

Complete Schema of Union Types It also joins all the union types, but relationships between entity types are also added. It is also a view of the complete U-Schema schema. This kind of schema corresponds to the logical schema typically used for relational databases.

7.2 Visualization of Complete Schemas

A simple *User Profile* database will be used as a running example throughout this paper. The database records data on users subscribed to a streaming service: personal data, watched movies and favorite movies; addresses will be separated from the rest of personal data. We will suppose that this database will be stored both in an aggregate-based system (e.g., MongoDB or Cassandra) and a graph system (e.g., Neo4j), and both stores will be called "UP-aggregate" and "UP-graph." Figure 7.1 shows the running example for the two stores.

In Figure 7.1, two *User* objects and one *Movie* object of "UP-aggregate" are shown in form of JSON documents stored in MongoDB. A *User* aggregates an *Address* object and an array of *WatchedMovies* objects, and also holds an array of references to *Movie* objects that records

the user's favorite movies. Each *WatchedMovies* holds a reference to the *Movie* watched by the user and the number of stars of the user's score. In the case of "UP-graph," *Addresses* and *WatchedMovies* are also connected to *Users* through reference relationships, as shown in Figure 7.1.

User and *Address* have two variations, while *Movie* and *WatchedMovies* have only one, i.e., there is not structural variability for these two entities. *User* and *Address* variations will be commented below when explaining the schema diagrams.

```
// User Collection
{
  _id: 178,
  name: "Brian",
  surname: "Caldwell",
  email: "brian_caldwell@gmail.com",
  address: {
    city: "Aylesbury",
    street: "Fairfax Cres",
    number: 6,
    postCode: 30760
  },
  watchedMovies: [
    {
      stars: 4,
      movie_id: 202
    }
  ],
  favoriteMovies: [
    202, 267, 378
  ]
}
```

```
// User Collection
{
  _id: 156,
  name: "Allison",
  email: "allison@gmail.com",
  address: {
    city: "Aylesbury",
    street: "Lott Meadow",
    number: 8
  },
  watchedMovies: [
    {
      stars: 3,
      movie_id: 202
    }, {
      stars: 5,
      movie_id: 295
    }
  ]
}
```

```
// Movie Collection
{
  _id: 202,
  name: "The Matrix",
  year: 1999,
  genre: "Science Fiction"
}
```

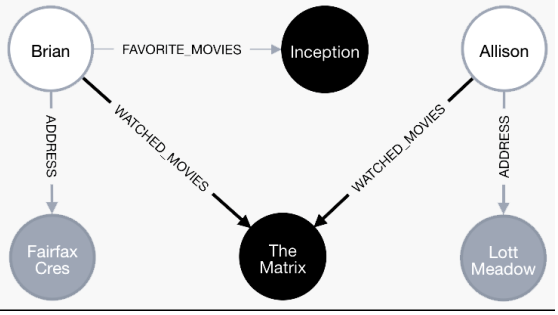

Figure 7.1: Running example for aggregate and graph stores.

Table 7.1 shows the mapping between U-Schema and graphical notation elements. Query results are visualized as diagrams in which there are two kinds of nodes:

- Schema types are represented as boxes with two compartments: «entity type» or «relationship type» stereotypes appear on the upper one, and the type name on the lower one; different colors are used to make it easier to identify the types of nodes: light yellow for root entity types, light gray for aggregate entity types, and light blue for relationship types.

- Variations are represented as white boxes with two compartments: the variation name and identifier appear in the upper one, and the list of features in the lower one.

These nodes are connected by means of four kinds of arrows as indicated in Table 7.1: (i) schema type to variation, (ii) variation to aggregated variation, (iii) variation to referenced entity, and (iv) reference to the relationship type that specifies it.

Features are prefixed with "+", "?", and "-" symbols to indicate if they are shared, non-shared, or specific. In the case of aggregation and reference arrows, this prefix is followed for the cardinality specification before the property name: "[0..1]" (zero to one), "[1..1]" (only one), "[0..*]" (zero to many), and "[1..*]" (one to many). It is worth noting that references and aggregations that belong to variations present in the query result but are not part of the set of relationships returned, will be shown in the lower compartment of its variation. They will appear in the same way as features, but indicating the kind of relationship ("--" or "<>-") and the cardinality.

Figures 7.2 and 7.3 show the U-Schema complete schemas extracted for "UP-aggregate" and "UP-graph", respectively. Both schemas can be obtained with the query "`FROM * TO *`", as discussed later in Section 7.3.2.

In Figure 7.2, the schema includes two root entity types: `User` and `Movie`, and two aggregated entity types: `Address` and `WatchedMovies` which are embedded into `User`.

`User` has two variations: `User[1]` only includes the shared attributes (`email`, `name`, and `_id`), while `User[2]` has the `surname` specific attribute and the `favoriteMovies` specific reference. Both variations aggregate `Address`, but each of them a different `Address`'s variation. `Address` has three shared properties: `city`, `number`, and `street`. Depending on whether the `postCode` optional property is present or not, two variations exist for `Address`.

In Figure 7.3, the schema includes three relationship types: `address`, `watchedMovies`, and `favoriteMovies`, and two entity types: the `User` and `Movie`. Relationship types

U-Schema Elements	Graphical Notation Elements
Entity Type	Box stereotyped with «Entity Type»
Relationship Type	Box stereotyped with «Relationship Type»
Structural Variation	Box with two compartments, one to specify the variation and another to include its list of feature specifications
Variation belongs to a schema type	Dashed arrow from an Entity box to a variation box
Attribute	Name and type separated by colon inside a variation box
Key	Prefix "Key" followed by the key's name and type inside a Variation box
Reference	Blue arrow directed from referencing entity to referenced entity, and labeled with the cardinality and reference name
Aggregation	Red arrow directed from aggregate entity to aggregated entity, and labeled with the cardinality and aggregation name, and decorated with a filled diamond at the aggregate class end
Variation featuring a reference	Dashed arrow from the middle of a reference arrow to a variation box of a relationship type

Table 7.1: Mapping between U-Schema and graphical notation.

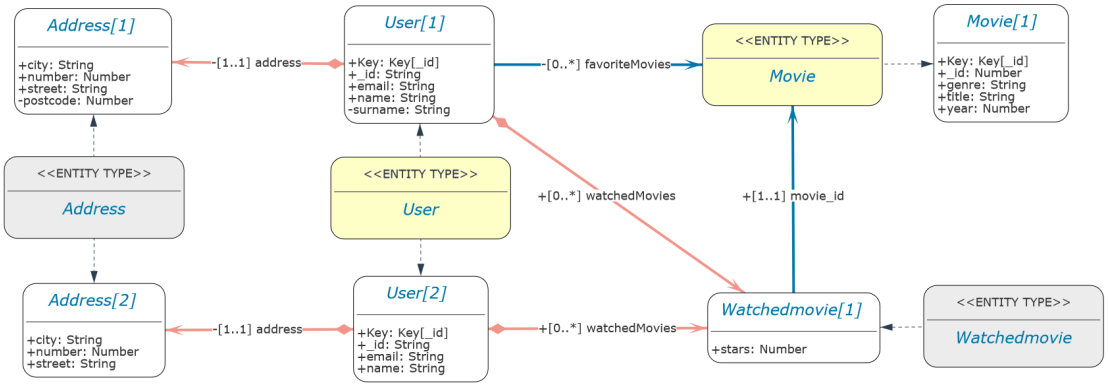

Figure 7.2: U-Schema complete schema for "UP-aggregate."

`address` and `watchedMovies` result from the aggregated entity types with the same name in the previous schema. As observed, each reference arrow is connected to the relationship

type that features it, e.g., the two existing `watchedMovies` references are connected to the only variation of the `watchedMovies` relationship type. Note that this type includes the attribute `stars`.

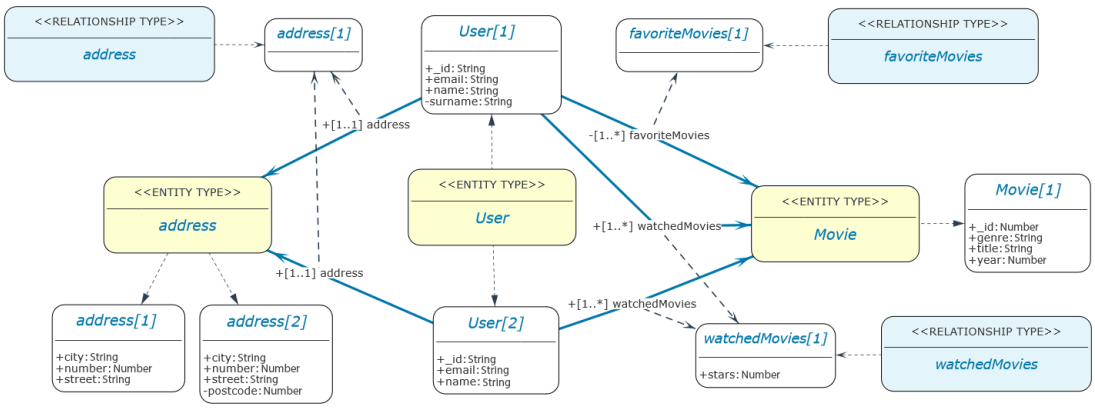

Figure 7.3: U-Schema complete schema for "UP-graph."

Finally, Figure 7.4 shows the complete schema of union types for "UP-aggregate", which would be obtained with the query `UNION FROM * TO *`.

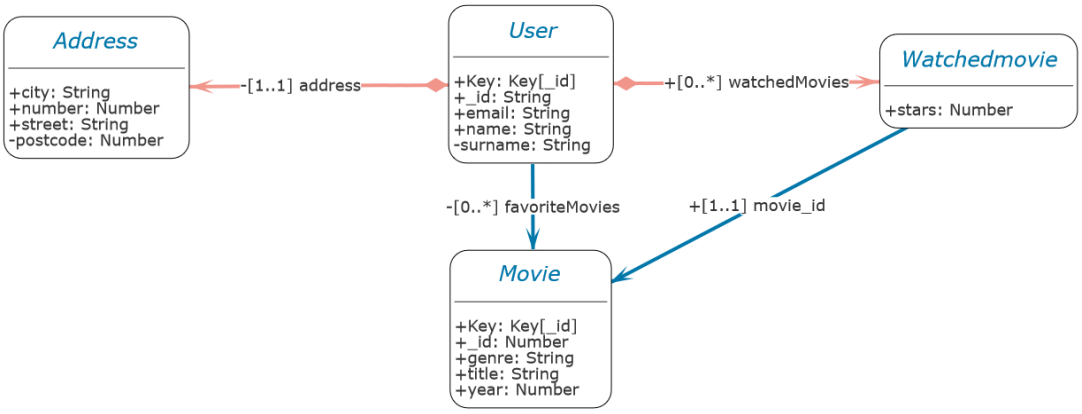

Figure 7.4: The complete schema of union types for "UP-aggregate" schema.

7.3 SkiQL Query Language: Syntax and Semantics

SkiQL was designed to be easy to learn, understand, and write. To achieve these characteristics, our choice was to create a command language and visualize the result of the queries

in form of a schema graph. Users should interactively write queries in a console with the results returned immediately. Being a command language encompasses other advantages, such as easily extending it with new query commands. SkiQL is intended for any stakeholder involved into the development of NoSQL database applications, such as database administrators, developers, and testers. We considered that these users could be interested in two kinds of queries: (i) recovering information on properties and variations of a particular entity type (and relationship type in the case of graph schemas), and (ii) checking the existing relationships (aggregations and references) among entity types. Both kinds of queries should return a sub-graph of the database schema. Also, the language should allow each previously defined schema and sub-schema type to be obtained. It is important to note that SkiQL is a DSL aimed to help database stakeholders to explore large database logical schemas, but it is not intended to express every possible query on a database schema.

To support the desired queries, SkiQL provides two kinds of declarative query statements: "query on one schema type," and "query on the path between schema types." Next, these query statements are described, and examples of queries on the `User Profile` schema will be shown to illustrate the application and usefulness of SkiQL.

7.3.1 Querying schema types

A "query on one schema type" (QT) allows the user to express a predicate on a schema type in order to extract information from its *type variations sub-schema*. More formally said: Given a schema S, a *schema type query qt* expresses an entity or relationship type specification *spec* that conveys a predicate $P(t)$ to be satisfied by the schema type t of S. Such specification consists of a partial intensional definition of t (a partial list of their features expressed with its name and optionally its data type), e.g., `User[name:string, favoriteMovies]`. If such a schema type t exists, the query returns the subgraph of the type variations subschema that satisfies the predicate. In this case, the relationships are enclosed in the lower compartment of the variations, as indicated above, and can be observed in Figure 7.5 that is commented below.

Regarding the syntax, a QT statement consists of three parts. First, a keyword indicating the schema type on which the query is applied: `ENTITY` for an entity type, `REL` for a relationship type, and `ANY` (both schema types). Next, the name of the schema type, which can be

followed of an optional *variation filter* clause. The type name can be expressed in different ways: the exact name to be matched, `*` to get all entities, use the symbol "`*`" to establish a prefix, suffix or both, e.g., `*Movie` to express an entity type name ending in "Movie", or Java-like regex expressions.

An excerpt of the EBNF grammar of this kind of query is the following:

```
<type-query> ::= ['UNION'] ('ENTITY' | 'REL' | 'ANY') <TypeSpec>
                 [<variation-filter>] | [<operations>]

<TypeSpec> ::= ['*']<typeName>['*']|'*'|'r"' <regexp> '"'

<variation-filter> ::= '[' <feature> {',' <feature>} ']'

<feature> ::= <nameFeature> ':' [<featureType>]

<featureType> ::= <AttributeType> | <AggregatedType> | <ReferenceType> | '?'

<AttributeType> ::= <BasicType> | <CollectionType>

<BasicType> ::= 'number' | 'string' | 'boolean'

<CollectionType> ::= <BasicType> '[' ']'

<AggregatedType> ::= 'AGGR' '<' <typeName> '>'

<ReferenceType>  ::= 'REF' '<' <typeName> '>'

<operations> ::= <operation> {',' <operation>}

<operation> ::=  'keys' | <date-interval>

<date-interval> ::=   'history' ('before' <date> | 'after' <date> | 'between'
   '('<date> ',' <date>')')
```

An *entity type variations subschema* or a *relationship type variation subschema* is returned when the `ENTITY` or `REL` keywords are followed by the name of an entity or relationship type, respectively. Figure 7.5 shows the results obtained for the query "`ENTITY User`" and Figure 7.6 for "`REL watchedMovies`" query. Note that two relationship type variation subschemas would be obtained for the query "`REL Movie`", those that corresponds to the `watchedMovies` and `favoriteMovies` relationship types. `ANY` keyword is used to express that the name can refer to either an entity type and a relationship type, e.g. this would occur if "`ANY Address`" is issued on the *User Profile* graph database. These three forms of query could be used to check if a schema type is present or not in the schema.

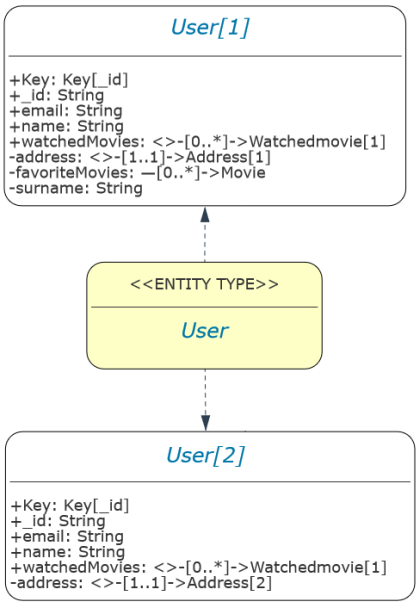

Figure 7.5: The `User` entity type variation subschema for "UP-aggregate."

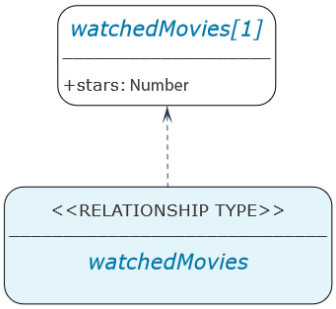

Figure 7.6: The `Watchedmovies` relationship type variation subschema for "UP-graph."

A *variation filter* enumerates the list of features that a variation must have in order to be selected, i.e. a QT query predicate. Each feature is specified by indicating its name and type separated by a colon, and features are separated by commas. The data types allowed for attribute features are `Number`, `String`, `Boolean`, as well as *collections* of values of these data types. The collections are those included in the U-Schema metamodel: Arrays, Sets, Lists, Tuples, and Maps. An array contains values of the same type, and the array type is specified by adding square bracket after the type name, for example `String[]`. The rest of collections are specified with the collection type name followed by the base type between angle brackets, for example, `Set<String>` and `Map<String, Number>`. A question mark can be used to indicate that the property type is unknown or either can be omitted. The types for relationship features are expressed with the prefix "AGGR" for aggregates or "REF" for references.

Query *Q1* shows a variation filter example: "find all `Users` variations with the `name: String` attribute, and another feature named `favoriteMovies` whose type is unknown." The result returned would be the same as Figure 7.5 but not including the variation `User[2]` that does not meet the filter.

Q1

```
ENTITY User [name: string, favoriteMovies]
```

A QT statement can also include operations that are applicable on schema types or variations. These operations follow the type name or filter. At this moment, two operations have been defined: "`keys`" returns the keys of the specified entity types, and "`history`" returns a graph that shows the timeline of appearance of variations in a given date interval.

In a variation filter, the `shared`, `non-shared`, and `specific` keywords can be used when specifying a feature. For example the query "`ENTITY * [shared id]`" would return all the entity types having a shared property named "`id`" of unknown data type, and "`ENTITY User [shared surname: string]`" would return all `User` variations having a shared property named `surname` of data type `String`. In the previous section, we showed the use of QT queries to obtain complete entity and relationship schemas: "`ENTITY *`" and "`REL *`".

7.3.2 Querying aggregations and references

As explained in Section 4, two kinds of relationships can be found in NoSQL stores: *aggregations* from a origin variation to another target variation (only in aggregate-based systems), and *references* from a origin variation to a target entity type. A *relationship query* (QR) selects the sub-schema that includes the specified relationships in the query. This kind of query can be formally defined as follows. Given a schema S, a *relationship query* qr expresses a specification oe of a entity type t belonging to S, and one or more relationship specifications $r_i, i = 1 \ldots n$, each of them indicating a relationship kind k_i and a specification of a target entity type tt_i. Thus, qr formulates a predicate $P(t, r)$ that is formed by a conjunction of logical operands, and each operand expresses that the relationship of kind k_i exists from t to tt_i. If this predicate is satisfied, the query returns the subgraph of S that contains the set of relationships r_i. While a QT query retrieves a subgraph of a type variations sub-schema, a QR query may return any subgraph of a complete schema.

Regarding the syntax, a QR query consists of a `FROM` clause followed by a `TO` clause. The former specifies the source type of the relationship, and the latter the target type and the kind of relationship. Depending on whether the prefix `UNION` is present or not, variations or union schema types are returned as source and target of the relationship returned.

The syntax is expressed below in form of an EBNF grammar.

```
<schema-query> ::= ['UNION'] <from-clause> <to-clause>

<from-clause> ::= 'FROM' (<entitySpec> [<variation-filter>] | '_')

<to-clause> ::= 'TO' <rel-spec> {',' <rel-spec>}

<rel-spec> ::= ['>>'] <entitySpec> [<variation-filter>]
                        ('REF' [<featureName>] [<variation-filter>] |
                         'AGGR' [<featureName>] | 'ANY' [<featureName>]) | '_'

<entitySpec> ::= ['*']<typeName>['*']|'*'|<regexp>)
```

A *from clause* is formed by the `FROM` keyword followed by an entity type name, and an optional variation filter that is expressed with the syntax exposed for QT queries. An empty `FROM` clause (underscore symbol) denotes that no entity type or variation could have a relationship to the target entity type.

A *to clause* is formed by a list of relationship specifications which are pairs formed by an entity type and a keyword denoting the kind of relationship: '`REF`", "`AGGR`", or "`ANY`". This latter keyword is used to indicate that the relationship can be aggregation or reference. A variation filter can only be used with `AGGR`. A star can be used to refer to "any entity type" and an underscore to "no entity type."

Q2 is a QR query specifying the condition "aggregation between `User` variations including the attribute `surname` of type String and an `Address` variation." As shown in Figure 7.7, if the query is applied on the "UP-aggregate" database schema, the result is the `User` variation "`User[1]`", which appears connected to the `Address` variation "`User[1]`" through an aggregation relationship. Note that the `watchedMovies` aggregation or `favoriteMovies` reference are not shown in form of edge as they are not part of the set of relationships satisfying the query predicate, so they appear in the variation box as features.

Q2

```
FROM User[surname:string]
TO Address AGGR
```

Table 7.8 shows more QR query examples for the `User Profile` schema. *Q3* checks if the `User` entity type has incoming relationships, and would return a message indicating that `User` is not target type of any relationship. *Q4* checks if `User` has references to `Movie` and aggregations to `Address`, and would return the `User[1]` variation connected to `Movie` and

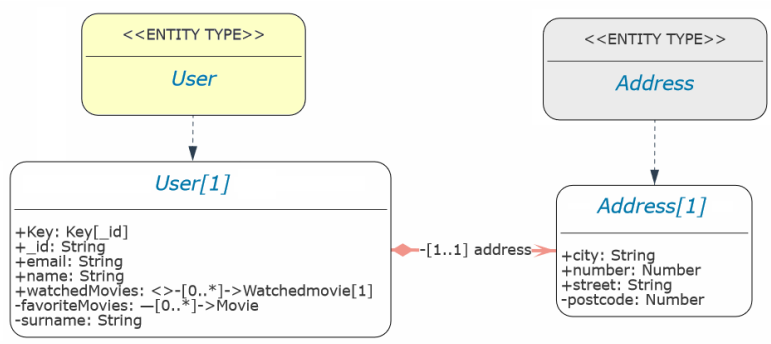

Figure 7.7: Subschema returned for queries Q2 and Q5 on "UP-aggregate" schema.

`Address` through the `favoriteMovies` reference and the `address` aggregation, respectively, as shown in Figure 7.9. Note that `watchedMovies` aggregation is not shown in form of an edge for the reason explained above.

Q3

```
FROM _
TO User
```

Q4

```
FROM User
TO Movie REF, Address AGGR
```

Q5

```
FROM User [favoriteMovies]
TO Address [postcode] AGGR
```

Q6

```
FROM User
TO >> Movie
```

Figure 7.8: QR query examples.

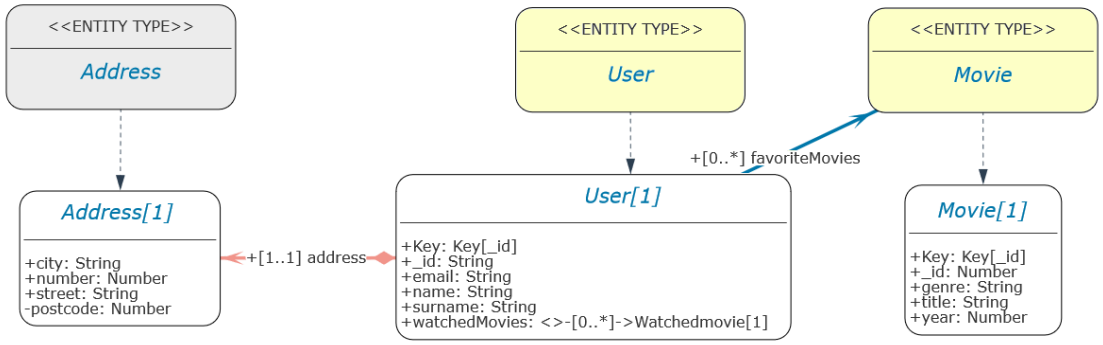

Figure 7.9: Subschema returned for query Q4 on "UP-aggregate" schema.

Q5 query retrieves relationships whose origin are `User` variations having `favoriteMovies` feature of unknown type, and the target is an `Address` variation containing the attribute

`postcode` through aggregation. The returned diagram is the same as for query *Q2*, which is shown in Figure 7.7.

In the query execution, relationships specified are direct by default, but a path of any length can be indicated by using the `>>` prefix, as illustrated in query *Q6*. This query checks if the `User` entity is connected to `Movie` by means of a path that can include any number of aggregations and references. Figure 7.10 shows the subschema returned where the `User[1]` variation is directly connected to `Movie` (`favoriteMovies` reference), and `User[2]` variation is indirectly connected to`Movie` through the `WatchedMovies` aggregate entity type that references `Movie`.

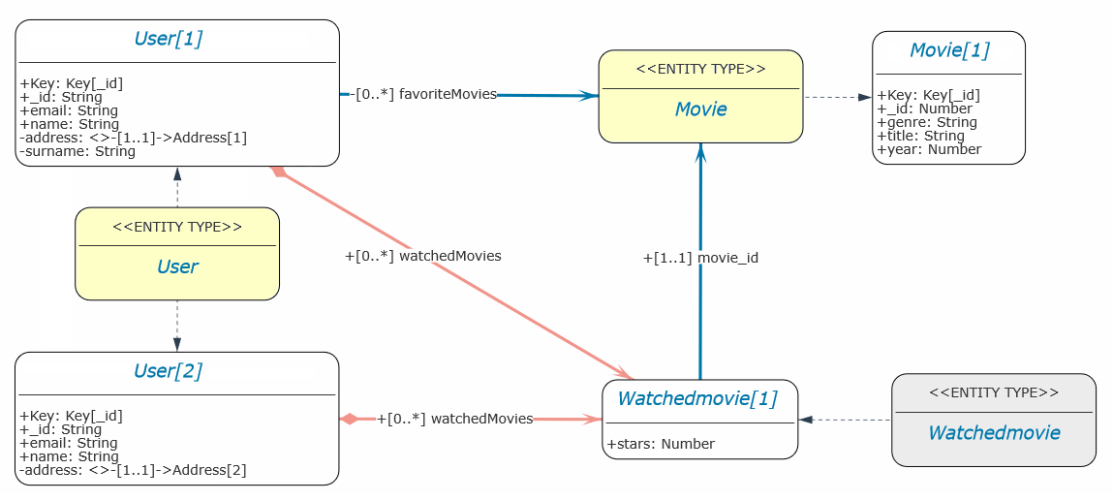

Figure 7.10: Subschema returned for query *Q6* on "UP-aggregate" schema.

All the relationships directly or indirectly incoming/outgoing to/from a given entity type can be obtained by using "`*`" to specify the schema type name, as shown below for the `User` entity type.

- `FROM User TO *` returns all relationships outgoing from `User`,
- `FROM * TO User` returns all relationships incoming to `User`.
- `FROM User TO >> *` returns all relationships outgoing from `User` to any entity type connected directly or indirectly.
- `FROM * >> TO User` returns all direct or indirect relationships incoming to `User`.

These four queries return union types instead variations if the `UNION` prefix is present.

In the `TO` clause, the keyword indicating the kind of relationship can be optionally followed by the name of a property, so that the query would only return relationships with that name. As references can have attributes in graph systems, they are instances of relationship types, SkiQL allows variation filters to be used to specify relationship attributes. Query *Q7* would check if `User` is connected to `Movie` through a reference which has a `stars` attribute of type `Number`. The result of this query issued on "UP-graph" schema is shown in Figure 7.11.

Q7

```
FROM User
TO Movie REF [stars: Number]
```

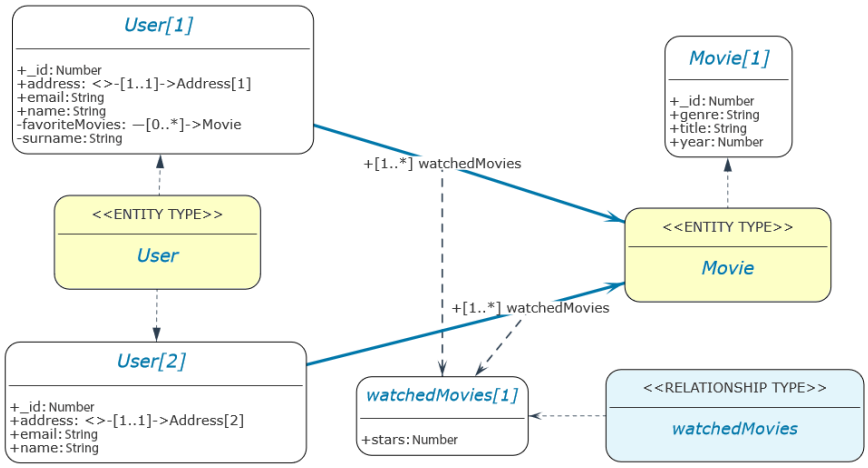

Figure 7.11: Subschema returned for query *Q7* on "UP-graph" schema.

Finally, *Q8* shows a QR query issued on "UP-graph" to find if the schema contains `User` entity type variations with the `surname` attribute, which are connected both to `Address` variations with `postcode` and to `Movie` through only `favoriteMovies` references. Figure 7.12 shows the result obtained for *Q8*.

Q8

```
FROM User [surname: string]
TO Address [postcode], Movie REF favoriteMovies
```

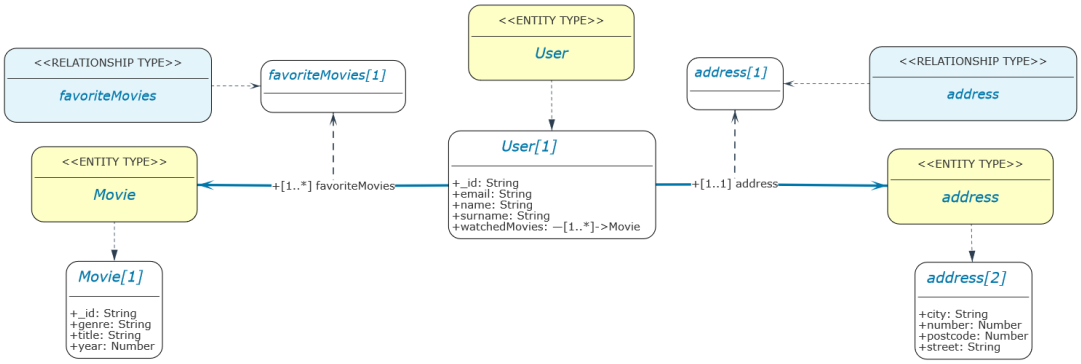

Figure 7.12: Subschema returned for query Q8 on "UP-graph" schema.

In addition to the graphical notation, the results can be also displayed as a set of tables, one for each returned schema type. Each table has a row for each variation, and rows have four columns: schema type name, variation identifier, number of instances, and a listing of features expressed in the format used in the diagrams. This textual notation can be useful if the number of variations returned is high.

7.4 Implementation of SkiQL

SkiQL was created with a metamodel-based language workbench, Xtext [22]. As it is well-known [117], these tools automate the building of DSLs by automatically generating an editor, a parser, and a model injector from the EBNF-like grammar or metamodel of the language.

Once the syntax of SkiQL was determined, a metamodel-based approach [61] was applied to implement the language. We first defined the metamodel (i.e., its abstract syntax) that can be seen in Figure 7.13, and then wrote the grammar in form of Xtext syntax rules. A translational approach [61] was applied to define the SkiQL semantics: SkiQL queries are written with the generated editor, and the model injector automatically produces SkiQL models in Ecore/EMF format [110]. Then, a query interpreter has as input these query models along with the U-Schema model that represents the NoSQL schema on which queries are issued.

The SkiQL metamodel, Figure 7.13, is divided into two parts, which correspond to the two types of query: *QT* on the right and *QR* on the left.

On the right, *RelationshipQuery*s (QT) can be specialized in *EntitySpec* to query entities, *RelationshipTypeSpec* to select `RelationshipType`s. *RelationshipQuery* is used for both. In addition, two kind of operation can be specified: to search keys with *KeysSpec* and also to search for `structural variaton` in a space of time using one of the classes that inherit from *Operation* and *VersionHistoryOperation* such as *All*, *After*, *Before*, and *Between*. Furthermore, every *SchemaSpec* can have a property filter, to search among the `Feature`s using *VariationFilter*, with which you can specify a set of properties with *PropertySpec* and any of the *Type*s as *PrimitiveType* or *RelationShipType*.

On the left, a *SchemaQuery*s (QR) can be defined of three types: (i) *RelationSpec* to select both aggregations and references; (ii) *AggregationSpec*, to select aggregations, and (iii) *ReferenceSpec* for references. With the relation `from` the origin entity is specified by means of an *EntitySpec*. In addition, *TargetExpression* specifies the target of the relationships, which can be another *SchemaQuery* (QR) or an *EntityExpression* to select an entity using an *EntitySpec*.

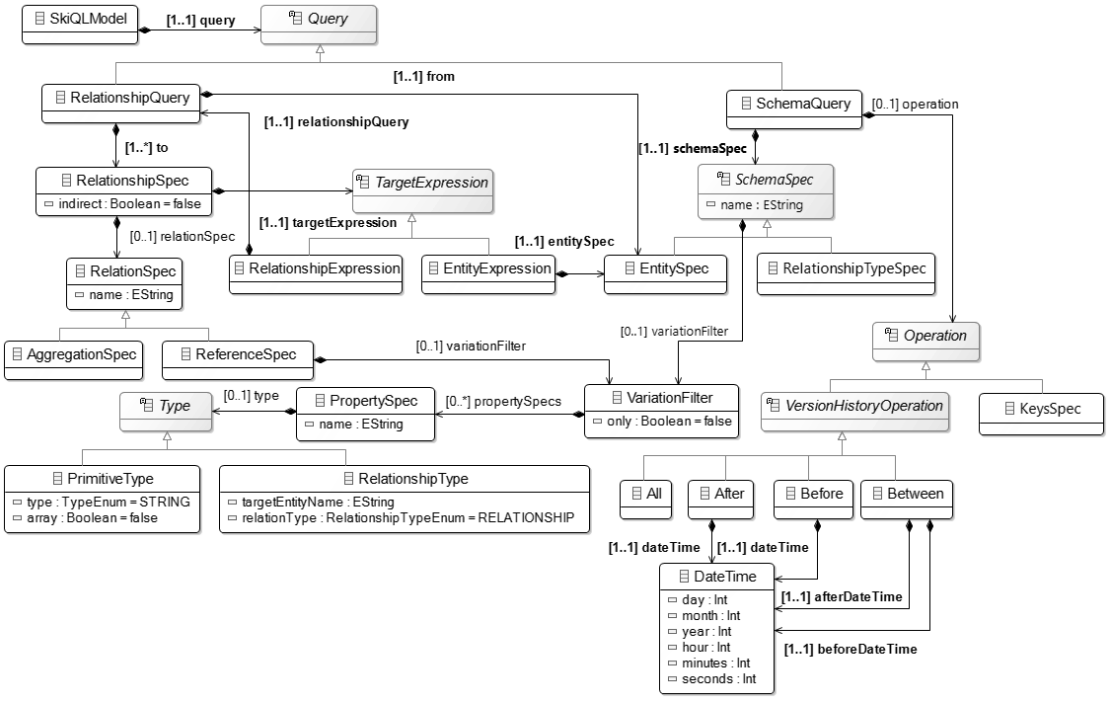

Figure 7.13: The SkiQL metamodel for the definition of the abstract syntax of the language.

Every time a QT or QR query is issued, the interpreter launches the injector execution to convert the query script into a SkiQL model. Then, the interpreter performs the following

two-step process. Firstly, the query model is analyzed to identify the conditions to be satisfied, and then the U-Schema model (i.e. the schema) is traversed to obtain the elements to be returned in the result graph. This second step is a model-to-model transformation that extract the part of the U-Schema model that constitute the desired result. Note that the U-Schema model is both the source and target metamodel in this transformation. The interpreter has been implemented with the language Xtend [22].

As shown in Figure 7.14, the interpreter has been integrated with a SkiQL schema viewer to graphically show the result of the SkiQL queries that are written in the console (i.e., the generated editor). This viewer receives as input the U-Schema model produced by the interpreter, and then creates the corresponding graph by applying the mapping exposed in Table 7.1 between U-Schema elements and graphical notation elements.

The viewer has been implemented by using VisJS* as graphical visualization API. Our tool (query interpreter and schema viewer) is a Web application that allows queries to be entered through an editor and also visualized in the browser.

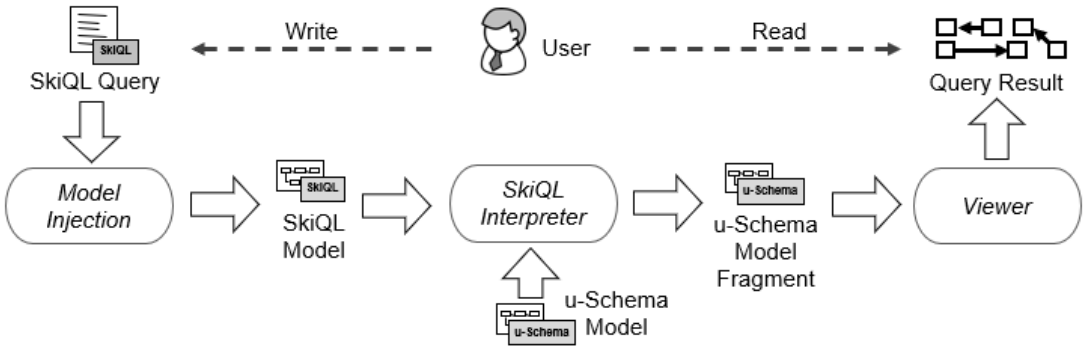

Figure 7.14: An overview of the visualization process and its implementation.

7.5 Evaluation

In this section, we will present the evaluation of SkiQL, which has been carried out in two forms. First, we measured some language metrics for SkiQL and compared the results with those of other query languages. Secondly, we surveyed some experienced NoSQL researchers on SkiQL features.

*VisJS Webpage: http://visjs.org.

Metric	Definition	SkiQL	GraphQL	Cypher	SPARQL
TERM	Terminals	29	44	115	158
VAR	Non-Terminals	39	71	99	123
HAL	Designer effort for grammar understanding	12.81	43.03	141.16	116.55
LRS	Complexity independent size	310	603	15756	15172
LAT/LRS	Ease of understanding	0.085	0.155	0.138	0.160

Table 7.2: SkiQL, GraphQL, Cypher and SPARQL metrics.

7.5.1 Calculating Language Metrics

In order to assess to what extent SkiQL is a simple and easy to learn language, we calculated the metrics defined in [41], and compared the results obtained to those of other three query languages, in particular: Cypher, SPARQL, and GraphQL. Table 7.2 shows the results for these four languages.[†]

The metrics used measure the following quantities. *TERM* and *VAR* the number of terminals and non-terminals, respectively. *HAL (Halstead metric)* the designer effort to understand the grammar. *LRS* the complexity of the language independent of its size, and *LAT/LRS* measures the ease of understanding the language.

Since queries were translated to Cypher in a first version of SkiQL, we have considered this graph query language. SPARQL and GraphQL were chosen as they are widely used query languages. The former is the standard RDF query language, and GraphQL is increasingly used to query APIs in web and mobile applications. The structure of GraphQL types is similar to U-Schema entity types but structural variations are not allowed. SQL was not considered because it is a large language whose specification has a large number of statements that are not used to query data.

Analyzing the results obtained for each language, we found that Cypher and SPARQL are larger languages than GraphQL and SkiQL, as they are intended to query more complex data structures. This is shown by the TERM and VAR values in Table 7.2. The TERM values are very close for SkiQL (29) and GraphQL (44), while Cypher (115) and SPARQL (158) have

[†]The `cfgMetrics` program was executed to calculate language metrics: https://code.google.com/archive/p/cfgmetrics/.

Listing 7.1: Cypher query for Q1: ENTITY User [name:string, favoriteMovies].

```
WITH ["name:string", "favoriteMovies"] AS properties
MATCH (p:Property) WHERE p.nameType IN properties
WITH collect(p) AS ps, properties
MATCH (ev:EntityVariation {entity:"Users"})-->(:Property)
WHERE ALL (p IN ps WHERE (ev -->(p))
RETURN ev
```

higher values. With non-terminals, the VAR value for SkiQL is significantly lower than the other three values. The metric HAL shows that Cypher is the more complex (141.2), followed by SPARQL (116.5), while GraphQL (43.0) and SkiQL (12.8) are much simpler languages. SkiQL is appreciably the least complex of the four. It is convenient to remark that Cypher is about ten times more complex than SkiQL. LRS is other metric that measures the language complexity, and its values are consistent with those obtained for HAL. Regarding LAT/LRS, Table 7.2 shows that GraphQL (0.160) and SPARQL (0.155) are somewhat more difficult to learn than Cypher (0.138), and the SkiQL value is half of the Cypher one. This difference in the easiness to learn a language is similar to those calculated for other DSLs defined as alternative to general purpose languages, such as [78]. SkiQL can be considered as a more abstract language defined on top of Cypher to query database schemas.

With SkiQL, developers save time writing queries with a simpler and both easier to understand and to learn language than Cypher to query schema graphs. The average number of LoC for simple queries expressed in Cypher is 8. This size is mainly due to the need of expressing the path to be traversed, and storing the visited nodes in variables. It should be noted that the size estimation was performed for simple queries. We then decided to build a domain-specific language (DSL) tailored to query schemas represented with the U-Schema unified metamodel. Listings 7.1 and 7.2 show Cypher queries for *Q1* and *Q4* SkiQL queries. Queries *Q2* and *Q8* are challenging to write in Cypher due to the difficulty of implementing variation filters. However these two queries are very easy to write in SkiQL. As shown in Listings 7.1 and 7.2, Cypher queries are longer and more complex than SkiQL queries. In addition, the user should learn Cypher and know how schemas are represented as graphs in the database.

Listing 7.2: Chyper query for Q4: FROM User TO Movie REF, Address AGGR.

```
MATCH c = allShortestPaths(
(:ENTITY {name:"Users"})-[:ENTITY_VARIATION|PROPERTY|REFS_TO*1..3]->
(:ENTITY {name:"Movies"}) )
MATCH c2 = allShortestPaths(
(:ENTITY {name:"Users"})-[:ENTITY_VARIATION|PROPERTY|AGGREGATES*1..3]->
(:ENTITY_VARIATION)<--(:ENTITY {name:"Address"}) )
RETURN c, c2
```

7.5.2 Survey on SkiQL Features

We surveyed a total number of 31 participants, which had no knowledge on SkiQL: 7 researchers from other research groups, 8 Spanish developers experienced in MongoDB, 6 members of our research group, and 10 students of a Big Data master.

We provided to the participants a document with three parts: a SkiQL tutorial with examples of queries and the result graph, several query exercises to be solved by respondents, and a questionnaire of six items to evaluate SkiQL. The questionnaire is shown in Table 7.3. Each question had to be assessed with a mark from 1 to 5 in the *Likert* scale.

The participants were provided with a virtual machine with all the necessary tools to write and execute SKiQL queries. Therefore, they all used the same environment and with the same assistants from the editor. They completed the questionnaire once they solved the exercises. The six questions asked were about the following features: legibility of result graphs, ease to learn, ease to understand queries, adequate expressiveness, usability of the environment, and usefulness of the language.

Results and Discussion

Table 7.3 shows the average and the standard deviation of the participant's scores for each of the six questions.

Easy to learn None of the respondents considered the language difficult to learn, the average obtained for this question is 4.19 with a standard deviation of 0.38, supporting SkiQL is easy to learn. In their comments, the participants indicated that the documentation provided had been very useful.

Usability This feature includes the ease of reading and understanding queries (second question of the survey) as well as the ease of writing queries (third question). The former

Question	AVG	SD
Is SkiQL easy to learn?	4.19	0.38
Is SkiQL easy to read?	4.84	0.35
Is SkiQL easy to write?	4.23	0.68
Is SkiQL expressiveness appropriate?	4.26	0.79
Could SkiQL be useful to developers?	3.81	0.72
Is the visualization understandable?	4.36	0.81

Table 7.3: Questionnaire results.

is the best evaluated in the survey, with an average of 4.84, and the participants strongly agreed that SkiQL is a simple and concise language. On the other hand, writing is well evaluated (4.23) but not so well as reading (4.84), although none of the participants scored negatively. Their scores divided equally between the two positive positions.

Expressiveness The expressiveness of the language refers to whether the language offers all the needed features. This characteristic is usually considered with the preciseness feature to measure the effectiveness: its ability to perform complex queries with the least number of elements. In our case, we only considered expressiveness because SkiQL is a command language, and a single query is executed each time. Most of the participants positively positioned on expressiveness with an average of 4.26.

Usefulness of language In the usefulness of the language, their scores divided equally between the neutral and positive answers (average is 3.81). More than half of respondents claimed to agree or strongly agree, and the other remaining took a neutral position. Some of them pointed out that the language had a great similarity in simplicity to the SQL language.

Legibility of the result graph The legibility of the graph returned refers to aspects such as the proper understanding of the schema in form of a graph, the ability to know the kind of each shown property as well as the variation which it belongs to, and understanding the relationships between different entities, among others. None of the participants considered that the visualization representation is illegible, and their scores divided equally between the neutral and positive positions. Thus most respondents are positively positioned regarding to the understanding of the graph that represent result schemas with an average of 4.36.

Limitations of the validation Among the limitations of the validation is that large schemas

were not used, in an attempt to reduce the effort to learn SkiQL. The survey included a limited number of exercises. We included very simple exercises to familiarize respondents with SkiQL, and then we wanted to assess the expressiveness of the language, and also to cover most of the features of the language, so we included more complex queries. This may have lead some participants to consider using the language to be slightly difficult. In the documentation given to respondents, we included a brief explanation on the metamodel used to represent schemas, and some of them had problems to solve exercises because they had not clear notions of structural variation of an schema type.

8

Conclusions and Future Work

The objectives of this thesis come from the three following ideas about the future of the databases.

"One Size does not fit all" Although relational systems will continue to prevail, at least, in this decade, NoSQL and new relational systems (New SQL) will be increasingly used. Relational systems could be adequate for traditional applications, New SQL for modern applications requiring high scalability and availability, and NoSQL when data is complex and its structure frequently changes. Today, the idea of "One Size does not fit all" [111] is widely accepted in the database academic and industry community. In fact, the database engine ranking shows several different paradigms or data models in the top 10: relational, document, spatial, graph, and RDF, and the first 8 database systems are multi-model, i.e., they support several data models. The popularity ranking therefore evidences the changes is happening in the database scope: the same company can use different database data models depending on the kind of application, and data requirements of modern applications can involve several data models to be satisfied.

Generic Database Tools are required Obviously, existing relational database tools are evolving to support emerging data paradigms, in particular the four kinds of NoSQL databases.

Also, new tools are providing services for different data models. That is, the challenge of database tools is to be generic instead of being tied to a particular data paradigm. This is happening with, for example, data modeling tool (e.g., ERStudio, Erwin, or Hackolade) or data query language (e.g., PartiQL or OrientDB-SQL). For this aim, a unified metamodel may be very useful as indicated in [118].

Schema extraction is essential for NoSQL systems The more attractive feature of NoSQL systems for developers is probably to be "schemaless". However, schemas are as essential as they are for relational systems: Developers must always design how data are logically and physically organized, and information recorded in the schemas is required to implement most of database tools. Therefore, developers should be assisted by schema management tools, and tools should include schema discoverers, which either can extract schemas from data or code.

In this thesis, we have addressed the main problems that arise in the development of generic database tools that integrate the most relevant data models, namely, relational and NoSQL models. Firstly, the definition of a unified metamodel that integrates relational and NoSQL data models. Secondly, the construction of logical schema extractors for each considered data model. Because the most schema extraction approaches have applied data analysis on document stores, we have investigated the code static analysis as an alternative. Thirdly, around the unified metamodel and the set of extractors, we have built a generic schema management tool that includes a schema query language and a schema graphical viewer. Tackling these issues, we faced to the challenges posed by a proposal of NoSQL logical schema that includes structural variations and the most common relationships between database entities.

Next, we will discuss to what extent the goals of the thesis, which were exposed in the Chapter 1, have been achieved.

8.1 Discussion

In Section 1.2, we defined the following seven goals:

Goal 1: The creation of a unified metamodel able to represent logical schemas: U-Schema.

Goal 2: The definition of the forward and reverse mappings between U-Schema and the data models that it integrates.

Goal 3: The design and implementation of a common strategy to extract U-Schema models from the data of different databases.

Goal 4: The implementation of an application code analysis approach to extract schemas and perform database refactoring.

Goal 5: The design of a generic schema query language of U-Schema models (SkiQL).

Goal 6: The implementation of a visualization notation of U-Schema models.

Goal 7: Explore the usefulness of U-Schema to define a database query language.

Next, we shall discuss the level of achievement for each of these goals.

8.1.1 Goal 1. Create a unified metamodel able to represent logical schemas: U-Schema

As far as we know, the definition of a unified data model for NoSQL stores was first pointed out by Allen Wang [118] with the purpose of designing a new generic architecture for the ERwin modeling. Recently, Hackolade recently announced the interest to define a generic data model [4]. However, no implementation of unified data model able to integrate relational and NoSQL stores has been completed yet, to our knowledge. In this thesis, the U-Schema unified metamodel has been created for the representation of logical schemas of NoSQL and relational databases. The metamodel was explained in the Chapter 4. It is worth noting that *New SQL* systems may also be considered since that their data models are relational. U-Schema differs from other proposal of generic metamodels as follows:

- Provides support for representing the logical schema of the four most common kinds of NoSQL stores (document, graph, key-value, and wide column) and relational systems.
- Includes the notion of *structural variation* to represent the different data structures that can exist for the same entity.

- Represent relationship types whose instances are references between entities in graph databases. This entails that relationship types can also have *structural variation*.

- Represent four kinds of relationships between entity types: aggregates, references, graph relationships, and generalization.

We have validated U-Schema by injecting schemas from database stores of systems covering all considered the data models.

8.1.2 Goal 2. Establishing the bidirectional mappings between U-Schema and the different individual data models

Building a unified data model, it is necessary to establish the mapping from it to each individual data model, and in the opposite direction. In the case of U-Schema, we have introduced the notion of *canonical mapping* in order to define this bidirectional mapping. In a canonical mapping, there is a natural and direct correspondence between elements of a data model and U-Schema elements. We have formally defined canonical mappings from the relational and four NoSQL data models to U-Schema (*forward mappings*), and from U-Schema to the individual data models (*reverse mappings*). Those mappings was explained in the Chapter 5. We have also established the reverse mappings between elements when a canonical mapping does not exist. This occurs because U-Schema has elements for which there are not a direct correspondence to elements of a individual data model. For example, U-Schema can represent relationship types which can only appear in graph stores. Then, a reverse mapping from U-Schema to the document data model should specify how relationship types are mapped to elements of this target model, in such a way that the mapping is bidirectional. This reverse mappings are those applied in database migrations, in which U-Schema will play the role of pivot representation.

U-Schema and the canonical mappings have been validated by developing schema extractors for the most popular system of each kind of NoSQL systems, and MySQL for relational systems. In each extractor, the mapping specified have been implemented. To do this, we have created a common strategy to extract the schema from stores: a map-reduce operation obtains the schema of the individual model, and then the canonical mapping is applied to generate the U-Schema model. This common strategy was explained in detail

in the Section 5.1. The map-reduce operation is common for aggregate-based data systems. Real datasets have been used to validate each extractor, whose performance has been measured and evaluated. Unlike schema extractors published, the extractors developed in this thesis are defined for a generic metamodel, and structural variation and relationships are addressed.

8.1.3 Goals 3 and 4. Build a generic language to query schemas and graphically visualize schemas

We have created the SkiQL generic schema query language. With SkiQL, developers can express queries on schemas represented as U-Schema models. The language is very simple and easy to understand, and it includes two main types of constructs: queries to recover information on an entity or relationship type and queries to know how two entity types are connected. A graphical notation has been devised to show the query result. Sub-schemas returned are represented as graphs whose nodes are *structural variation* and the edges are U-Schema relationships. This visualization has been implemented as a Web application. The language and the graphical notation were described in detail in the Chapter 7.

Some language metrics [41] have been calculated to evaluate SkiQL, e.g. expressiveness, understandability and legibility. Also, a first version of SkiQL and the graphical notation were evaluated through a survey submitted to people having some kind of experience about NoSQL and relational databases. The former evaluation evidenced we had achieved the objectives desired for SkiQL. The latter served to improve the language and design a new visualization developed from scratch instead of using Neo4j tooling.

8.1.4 Goal 5 and 6. Design and implement a code analysis strategy to discover NoSQL schemas, and automate the " join query removal" refactoring

Code static analysis of NoSQL applications had only been addressed in an approach aimed to discover the evolution of MongoDB containers from different versions of the same Java application [86]. A few months ago, a analysis static work for finding Database Accesses in MongoDB Applications has been presented [33]. As explained in the Chapter 6, Our work of code static analysis goes further than these two approaches because we have developed a model-driven re-engineering solution that extract logical schemas, and use these

schemas to apply database refactorings. To achieve this, we have defined a metamodel able to represent the basic constructs of procedural and object-oriented languages, which has to be extended for each concrete programming language. Other original metamodel has been defined to represent the CRUD operations and data structures discovered in the code analysis (DOS metamodel). This latter model has been mapped to U-Schema to obtain the logical schema. Then, the schema and the reverse engineered models can be analyzed to apply automated database refactorings. In the thesis, we have automated the "join query removal" refactoring in order to improve the performance of the code. Having different versions of the application, our approach also allows the schema evolution to be obtained as in [86]. Code analysis is useful to find references between entities. While join queries allow references to be exactly identified in the code analysis, they can only be inferred from data by using some heuristics, but these heuristics cannot guarantee that all references are identified.

We have validated our code analysis approach through a round-trip process. We defined a database schema for a MongoDB store, and we created an U-Schema model. Then, we made a series of changes on this schema to perform a manual database refactoring that produced a refactored U-Schema model. After that, we implemented a Javascript application that uses the schema. Finally, we applied the code analysis process to the application and we compared the original U-Schema model with the model obtained in the code analysis We have also validated the refactoring by comparing the manually refactored U-Schema models with the model obtained by processing the stored data.

8.1.5 Goal 7. Explore the usefulness of U-Schema to define a database query language

We have conducted a study to create a universal query language based on U-Schema. As we explained in Section 4.5.1, we identified the type of queries that would be necessary and have proposed an SQL-like syntax. For the different kinds of queries, we have also illustrated some query examples. Finally, we specified a possible architecture needed to implement the execution of the universal query language.

8.2 Contributions

In this section, we shall indicate the main research contributions of this thesis.

To our knowledge, this thesis contributes with the first logical unified metamodel that integrates the most widely used database paradigms: Relational and NoSQL. The creation of this metamodel entailed other contributions:

- The definition of two logical data models for NoSQL systems: one for aggregate-based systems (columnar, document, and key-value) and one for graph systems. These data models are more complex than those previously published because they include different kinds of relationship as well as structural variation.
- The formal specification of the bidirectional mappings between the unified metamodel and the individual data models.
- The definition of an architecture with reusable components to create schema extractor from any NoSQL system. The results of the work related to the definition of the unified metamodel, the specification of mappings, and the implementation of extractors, have been presented in a long paper of 26 pages published in the Information Systems Journal [29].

Our unified metamodel represents logical database schemas, but sometimes it is necessary to know information on physical aspects such as those related to indexes or data partitioning. For this reason, the results of this thesis allowed us to define the first approach addressing the connection between logical and physical schemas. The results obtained were published in the workshop CoMoNoS (*Conceptual Modeling for NoSQL data stores*) that is part of the Conceptual Modeling Conference since 2020 [93]. In that paper, we present the mappings between the U-Schema metamodel and a physical metamodel created for MongoDB.

Regarding our code static analysis work, we contribute with the first proposal of logical schema extraction from code. When this thesis started, only one work on code static analysis for NoSQL applications had been published by Meurice and Cleve [86]. This was limited to extract the union schema of each MongoDB collection, but relationships and structural variation were not addressed. Our code analysis proposal was made by an collaboration with Dr Cleve in a predoctoral stay in the PRECISE group (University of Namur, Belgium). Beyond the extraction of the logical schema, our code analysis strategy also addressed the automation of schema changes such as refactorings. An infrastructure has been created

to facilitate the detection of the source code lines to be modified. We have applied this infrastructure to implement the "join queries removal" for JavaScript applications accessing to MongoDB stores. The code and DOS models are useful to detect the code smell for NoSQL applications identified in [19]. To our knowledge, our proposal is the first work about automating NoSQL schema changes. However, the great effort devoted to this work and elaborating the paper has made it difficult to write the papers presenting the results related to our code analysis. At this moment, we are preparing two publications. A first paper that describes how schemas are obtained from code and how data and code analysis are needed to extract the correct schema. In a second paper, we present our code infrastructure to automate schema evolution and its application to the "join queries removal refactoring."

In this thesis, we have defined the first proposal of a generic NoSQL schema query language. Moreover, the SkiQL language is generic because it has been designed for U-Schema. SkiQL allows developers to express queries on logical schemas represented as U-Schema models. A paper describing SkiQL and the graphical interactive visualization of query results has been submitted to Data & Knowledge Engineering, and it is available in arxiv [53]. Before creating SkiQL, we experimented with the *Cypher* language to query U-Schema models injected into Neo4j stores, and the *Neo4j Browser tool* was used to visualize the query results (i.e., graphs). We published this proof of concept in the Spanish Conference of Software Engineering and Databases of 2019 [54].

Finally, other original contributions are:

- A study on the usefulness of U-Schema to create a generic language to query NoSQL stores of any kind of system. We identified the types of queries that would be needed, and a syntax was proposed, as discussed in [29].

- A comparison of the different generic metamodels proposed to represent database schemas or data formats. This analysis was presented in Chapter 3.

- MDE has been applied to implement all the approaches devised in this thesis: unified metamodel and bidirectional mappings (i.e., schema extractors), the SkiQL DSL, and the reverse engineering process for the code analysis. Therefore, a comparison of model-to-model transformation languages was carried out to choose the more con-

venient solution for the thesis. This study was presented in the Spanish Conference of Software Engineering and Databases of 2018 [52].

- The use and validation of the MDE development methodology designed by the author of this thesis [28]. This methodology has been applied to develop the code analysis approach of the thesis.

8.3 Future Work

The research work carried out enables to continue with the construction of generic tooling by taking advantage of the unified metamodel. Among these future works, we would remark the following:

- To define physical schema metamodels and investigate the mappings from/to U-Schema.
- A U-Schema-based Generic Query Language to query data from different kind of stores.
- To implement database migration by using U-Schema and the established mappings.
- To create a data mapping language to express specialized mappings between two NoSQL data models.
- To build an agile database approach and tool aimed to favor an agile development with databases.

Before commenting these works, it is convenient noting that U-Schema has been used in another thesis of our research group that started when the metamodel had already been defined [76]. In that thesis, Alberto Hernández Chillón has addressed the creation of a DSL family: a universal schema definition language, a schema changes generic language, and a language for the generation of datasets for testing purposes.

Physical metamodel and mappings with logical U-Schema The metamodel presented concerns to the logical view, and new metamodels could represent physical schemas as well as a unified physical schema. Thus, we will have U-Schema-Physical and U-Schema-Logical,

where physical schemas will be extracted from data stores, and logical schemas could be directly obtained either from stores or from physical schemas, as described in [93]. Physical data models for each system will include data structures at physical abstraction level, indexes, physical data distribution, among others. Regarding improvements of U-Schema, we will extend the metamodel to represent constraints to support new logical validation characteristics in some NoSQL databases, such as the MongoDB Schema Validation.

A U-Schema-based Generic Data Query Language Given the widespread usage of different data models, developers and companies face the problem of managing several data query languages. Therefore, there exists a great interest in creating a universal query language for the variety of data managed in modern applications, and some proposals have recently appeared. We have made a study to identify the type of queries would be needed and proposed a syntax but we have not implemented the language.

Database Migrations Database migration is a typical task in which a unified or generic representation provides a great advantage. The migration process can be simplified by using U-Schema models. The U-Schema models from the source database can be used to automatically generate queries to read all the data in the original database, to classify the data into structural variations and perform some processing applying some migration rules. The migration rules could be hardcoded, or either specified with a language. This language would be defined taking into account the abstractions of U-Schema.

The results achieved in this thesis along with those of the Hernandez's thesis are the basis of a new research project of the ModelUM group, which has received funds of the Spanish Ministry of Science, Innovation and Universities.* This project is tackling the definition of an agile approach to evolve database schemas. Recently, this topic has received great attention for relational databases, and several popular tools are available as Liquibase† or Flyway.‡ In our project, we are using U-Schema to develop a generic tool, and the schema change scripts are written with the Orion language defined in the Hernández's thesis [76]. Once changes are applied on the schema, the code infrastructure defined in the thesis here

*Project Identifier: PID2020-117391GB-I00.
†Liquidbase website: https://www.liquibase.org/.
‡Flyway website: https://flywaydb.org/.

presented will be used to update data and code. Also data migration will be addressed by considering the mappings established in this thesis.

8.4 Publications, Projects, and Grants

In this section we note all the articles and conference papers developed during the work of the thesis duration along with the projects involved and grants received.

Articles

- Carlos Javier Fernández Candel, Diego Sevilla Ruiz, and Jesus García Molina. *A Unified Metamodel for NoSQL and Relational Databases*. Information Systems, 2021. 10.1016/j.is.2021.101898
- Carlos Javier Fernández Candel, Jesús García Molina and Diego Sevilla Ruiz. *SkiQL: A Unified Schema Query Language*. Data & Knowledge Engineering. (Submitted). arxiv.org/abs/2204.06670
- Carlos Javier Fernández Candel, Jesús García Molina, Francisco Javier Bermudez Ruiz, José Ramón Hoyos Barceló, Diego Sevilla Ruiz, and Benito José Cuesta Viera. *Developing a model-driven reengineering approach for migrating PL/SQL triggers to java: A practical experience*. Journal Systems and Software, 151:38–64, 2019. 10.1016/j.jss.2019.01.068
- Carlos Javier Fernández Candel, Jesús García Molina, Diego Sevilla Ruiz and Anthony Cleve. *NoSQl Schema Extraction from Application code*. (On development).
- Carlos Javier Fernández Candel, Jesús García Molina, Diego Sevilla Ruiz and Anthony Cleve. *NoSQL Refactoring to improve performance*. (On development).

International Conferences

- Pablo David Muñoz Sánchez, Carlos Javier Fernández Candel, Jesús García Molina, and Diego Sevilla Ruiz. *Extracting Physical and Logical Schemas for Document Stores*. CoMoNoS Workshop in Conceptual Modeling International Conference, 2020. 10.1007/978-3-030-65847-2_15

National Conferences

- Carlos Javier Fernández Candel, Diego Sevilla Ruiz, and Jesús García Molina. *Utilización de Neo4j para consultar esquemas de Bases de Datos NoSQL*. XXIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Cáceres, Spain, September 2019. hdl.handle.net/11705/JISBD/2019/080

- Carlos Javier Fernández Candel, Jesús García Molina, Francisco Javier Bermúdez Ruiz, and Diego Sevilla Ruiz. *Una experiencia con transformaciones modelo-modelo en un proyecto de modernización*. XXIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Sevilla, Spain, September 2018. hdl.handle.net/11705/JISBD/2018/039

Projects

- *A Model-Based Environment to Support NoSQL Data Engineering*. Ayudas a proyectos de Investigación y Desarrollo. Issued by the Spanish Ministry of Science, Innovation and Universities.
  Project grant: TIN2017-86853-P, 2017.

- *An Agile Development Approach for the NoSQL Database Schema Evolution: Data and Code Migration*. Ayudas a proyectos de Investigación y Desarrollo. Issued by the Spanish Ministry of Science, Innovation and Universities.
  Project grant: PID2020-117391GB-I00, 2021.

Stay

- *Stay at the group Research Center in Information System Engineering (PRECISE)* [§], The Faculty of Computer Science and the Department of Business Administration, Université de Namur, Belgium, 2019.
  Under the grant EST18/00760.

[§]PRECISE website: https://www.unamur.be/en/precise

Grants

- *Ayudas para la Formación de Profesorado Universitario (FPU)*.
  Issued by the Spanish Ministry of Science, Innovation and Universities.
  Grant Number: FPU16/02203, 2016.

- *Ayudas a la Movilidad para Estancias Breves y Traslados Temporales*.
  Issued by the Spanish Ministry of Science, Innovation and Universities.
  Grant Number: EST18/00760, 2018.

References

[1] Acceleo Webpage. Accessed May 2022. URL: http://www.eclipse.org/acceleo/.

[2] Announcing PartiQL: One query language for all your data. Accessed: May 2022. URL: https://aws.amazon.com/es/blogs/opensource/announcing-partiql-one-query-language-for-all-your-data/.

[3] Dataedo Webpage: Database Data Dictionary Query Toolbox. Accessed: May 2022. URL: https://dataedo.com/kb/query.

[4] Hackolade - HackPolyglot Data Modeling. Accessed: May 2022. URL: https://hackolade.com/polyglot-data-modeling.html.

[5] Neo4j browser. Accessed: May 2022. URL: https://neo4j.com/developer/neo4j-browser/.

[6] OrientDB SQL Reference. Accessed: May 2022. URL: http://orientdb.com/docs/3.1.x/sql/.

[7] Partiql specification. Accessed: May 2022. URL: https://partiql.org/assets/PartiQL-Specification.pdf.

[8] Querying metadata. Accessed: May 2022. URL: http://orientdb.com/docs/3.1.x/sql/SQL-Metadata.html.

[9] Xtend Webpage. Accessed: May 2022. URL: http://www.eclipse.org/xtend/.

[10] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[11] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[12] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017. doi:10.1145/3104031.

[13] Nicolas Anquetil, Anne Etien, Mahugnon H Houekpetodji, Benoit Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatiha Djareddir, Jerôme Sudich, and Mustapha Derras. Modular moose: A new generation software reverse engineering environment. *arXiv preprint arXiv:2011.10975*, 2020.

[14] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the NoSQL world. *Computer Standards & Interfaces*, 67, 2020. doi:10.1016/j.csi.2016.10.003.

[15] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform Access to Non-relational Database Systems: The SOS Platform. In *24th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 160–174, Gdansk, Poland, June 2012. doi:10.1007/978-3-642-31095-9_11.

[16] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to NoSQL systems. *Information Systems*, 43:117–133, 2014. doi:10.1016/j.is.2013.05.002.

[17] Paolo Atzeni, Giorgio Gianforme, and Paolo Cappellari. A universal metamodel and its dictionary. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 1:38–62, 2009. doi:10.1007/978-3-642-03722-1_2.

[18] Vladimir Bacvanski and Charles Roe. Insights into NoSQL Modeling: A Dataversity Report, 2015.

[19] Jehan Bernard and Thomas Kintziger. MongoDB Code Smells: Defining, Classifying and Detecting Code Smells for MongoDB Interactions in Java Programs. Master's thesis, Faculté d'informatique. Université de Namur, Belgium, 2021.

[20] Philip A. Bernstein, Alon Y. Halevy, and Rachel Pottinger. A vision of management of complex models. *SIGMOD Rec.*, 29(4):55–63, 2000. doi:10.1145/369275.369289.

[21] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1–12. ACM, 2007. doi:10.1145/1247480.1247482.

[22] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016.

[23] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[24] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. MoDisco: A Model Driven Reverse Engineering Framework. *Information & Software Technology*, 56(8):1012–1032, 2014. doi:10.1016/j.infsof.2014.04.007.

[25] Hugo Brunelière, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. pages 173–174, 2010. doi:10.1145/1858996.1859032.

[26] Nicolas Bruno and Surajit Chaudhuri. Flexible Database Generators. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1097–1107.

[27] Peter Buneman. Semistructured Data. In *Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, Arizona, USA, 1997. ACM. doi:10.1145/263661.263675.

[28] Carlos Javier Fernández Candel, Jesús García Molina, Francisco Javier Bermudez Ruiz, José Ramón Hoyos Barceló, Diego Sevilla Ruiz, and Benito José Cuesta Viera. Developing a model-driven reengineering approach for migrating PL/SQL triggers to java: A practical experience. *Journal Systems and Software*, 151:38–64, 2019. doi:10.1016/j.jss.2019.01.068.

[29] Carlos Javier Fernández Candel, Diego Sevilla Ruiz, and Jesús Joaquín García Molina. A unified metamodel for nosql and relational databases. *Information Systems*, 104:101898, 2022. doi:10.1016/j.is.2021.101898.

[30] Cassandra webpage. Accessed: May 2022. URL: http://cassandra.apache.org/.

[31] R. G. G. Cattell and Douglas K. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[32] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. doi:10.1145/320434.320440.

[33] Boris Cherry, Pol Benats, Maxime Gobert Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of database accesses in mongodb applications. In *IEEE 29th International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, Hawaii, USA, March 15-18, 2022*.

[34] Yun Chi, Yirong Yang, and Richard R. Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowledge and Information Systems*, 8(2):203–234, 2005. doi:10.1007/s10115-004-0180-7.

[35] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Athena: A database-independent schema definition language. In *Advances in Conceptual Modeling - ER 2021 Workshops CoMoNoS, EmpER, CMLS, St. John's, NL, Canada, October 18-21, 2021, Proceedings*, volume 13012, pages 33–42. Springer. doi:10.1007/978-3-030-88358-4_4.

[36] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Towards a taxonomy of schema changes for nosql databases: The orion language. In *Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings*, volume 13011, pages 176–185. Springer, 2021. doi:10.1007/978-3-030-89022-3_15.

[37] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970. doi:10.1145/362384.362685.

[38] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Typing massive JSON datasets. In *International Workshop on Cross-model Language Design and Implementation*, volume 541, pages 12–15, 2012.

[39] Isabelle Comyn-Wattiau and Jacky Akoka. Model driven reverse engineering of nosql property graph databases: The case of neo4j. In *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*, pages 453–458. IEEE Computer Society, 2017. doi:10.1109/BigData.2017.8257957.

[40] Rick Copeland. *MongoDB Applied Design Patterns*. O'Reilly Media, Inc., 2013.

[41] Matej Crepinsek, Tomaz Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, and Gilles Roussel. On automata and language based grammar metrics. *Computer Science and Information Systems*, 7(2):309–329, 2010. doi:10.2298/CSIS1002309C.

[42] Cypher introduction. Accessed: May 2022. URL: https://neo4j.com/docs/cypher-manual/current/introduction/.

[43] Dbschema webpage. Accessed: May 2022. URL: http://www.dbschema.com.

[44] Julien Delplanque, Anne Etien, Olivier Auverlot, Tom Mens, Nicolas Anquetil, and Stéphane Ducasse. Codecritics applied to database schema: Challenges and first results. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 432–436. IEEE Computer Society, 2017. doi:10.1109/SANER.2017.7884648.

[45] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal? UML shortcomings for coping with round-trip engineering. In *guillemotleftUML-guillemotright'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723, pages 630–644. Springer, 1999. doi:10.1007/3-540-46852-8_44.

[46] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. Mse and famix 3.0: an interexchange format and source code model family. 2011.

[47] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[48] Vincent Englebert and Jean-Luc Hainaut. DB-MAIN: A next generation meta-case. *Information Systems*, 24(2):99–112, 1999. doi:10.1016/S0306-4379(99)00007-1.

[49] ER-Studio Webpage. Accessed: May 2022. URL: https://www.idera.com/er-studio-enterprise-data-modeling-and-architecture-tools.

[50] ERwin Data Modeler Webpage. Accessed: May 2022. URL: http://erwin.com/products/erwin-data-modeler.

[51] Severino Feliciano. *Inferring NoSQL Data Schemas with Model-Driven Engineering Techniques*. PhD thesis, Faculty of Informatics. University of Murcia, Spain, 2017.

[52] Carlos Fernández Candel, Jesús García Molina, Francisco Javier Bermúdez Ruiz, and Diego Sevilla Ruiz. Una experiencia con transformaciones modelo-modelo en un proyecto de modernización. In *XXIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, Sevilla, Spain, September 2018. URL: hdl.handle.net/11705/JISBD/2018/039.

[53] Carlos Fernández Candel, Jesús García Molina, and Diego Sevilla Ruiz. Skiql: A unified schema query language. *arXiv preprint arXiv:2204.06670*, 2022. URL: arxiv.org/abs/2204.06670.

[54] Carlos Fernández Candel, Diego Sevilla Ruiz, and Jesús García Molina. Utilización de Neo4j para consultar esquemas de bases de datos NoSQL. In *XXIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, Cáceres, Spain, September 2019. URL: hdl.handle.net/11705/JISBD/2019/080.

[55] Jérôme Fink, Maxime Gobert, and Anthony Cleve. Adapting queries to database schema changes in hybrid polystores. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September, 2020*, pages 127–131. IEEE. doi:10.1109/SCAM51674.2020.00019.

[56] Apache Foundation. Apache drill. Accessed: May 2022. URL: http://drill.apache.org/.

[57] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[58] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[59] Martin Gogolla. *An extended entity-relationship model: fundamentals and pragmatics*. Springer, 1994.

[60] Stefan Gössner. JSONPath – XPath for JSON, July 2020. URL: https://tools.ietf.org/id/draft-goessner-dispatch-jsonpath-00.html.

[61] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley, 2004.

[62] Object Management Group. AFP specification. http://www.omg.org/spec/AFP/1.0/. Accessed: May 2022.

[63] Object Management Group. ASTM specification. http://www.omg.org/spec/ASTM/1.0/. Accessed: May 2022.

[64] Object Management Group. Meta Object Facility (mof) 2.0 Query/View/Transformation (qvt). http://www.omg.org/spec/QVT/. Accessed: May 2022.

[65] Object Management Group. Object constraint language (ocl) webpage. http://www.omg.org/spec/OCL/. Accessed: May 2022.

[66] Object Management Group. SMM specification. http://www.omg.org/spec/SMM/1.1.1/. Accessed: May 2022.

[67] Object Management Group. Architecture-Driven Modernization (ADM). http://adm.omg.org/, 2007.

[68] Object Management Group. *Knowledge Discovery Meta-Model (KDM)*, 2016. Document formal/2016-09-01. URL: https://www.omg.org/spec/KDM/1.4/.

[69] Hackolade Webpage. Accessed: May 2022. URL: https://hackolade.com/.

[70] Jean-Luc Hainaut. A generic entity-relationship model. In *Information System Concepts: An In-depth Analysis, Proceedings of the IFIP TC8/WG8.1 Working Conference on*

Information System Concepts: An In-depth Analysis (ISCO 1989), Namur, Belgium, October, 1989, pages 109–156. North-Holland, 1989.

[71] Jean-Luc Hainaut. The transformational approach to database engineering. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, pages 95–143, 2005. doi:10.1007/11877028_4.

[72] Jean-Luc Hainaut, Vincent Englebert, Jean Henrard, Jean-Marc Hick, and Didier Roland. Database Evolution: the DB-Main Approach. In *Entity-Relationship Approach - ER'94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, UK, December, 1994*, volume 881, pages 112–131. Springer. doi:10.1007/3-540-58786-1_76.

[73] Hbase webpage. Accessed: May 2022. URL: https://hbase.apache.org/.

[74] Alberto Hernández Chillón, Severino Feliciano Morales, Diego Sevilla Ruiz, and Jesús García Molina. Exploring the Visualization of Schemas for Aggregate-Oriented NoSQL Databases. In *ER Forum 2017, 36th International Conference on Conceptual Modeling (ER)*, volume 1979, pages 72–85, Valencia, Spain, November 2017.

[75] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García-Molina. Deimos: A Model-based NoSQL Data Generation Language. In *Advances in Conceptual Modeling - ER 2020 Workshops CoMoNoS, Viena, Austria*, volume 12584, pages 151–161, 2020. doi:10.1007/978-3-030-65847-2_14.

[76] Alberto Hernández Chillón. *A Multi-Model Environment for NoSQL and Relational Systems*. PhD thesis, Faculty of Informatics. University of Murcia, Spain, 2022.

[77] Jean-Marc Hick and Jean-Luc Hainaut. Strategy for database application evolution: The DB-MAIN approach. In *Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October, 2003*, volume 2813, pages 291–306. Springer. doi:10.1007/978-3-540-39648-2_24.

[78] José Ramón Hoyos, Jesús García Molina, and Juan A. Botía. A domain-specific language for context modeling in context-aware systems. *Journal of Systems and Software*, 86(11):2890–2905, 2013. doi:10.1016/j.jss.2013.07.008.

[79] Ion's type system specification. https://amzn.github.io/ion-docs/docs/spec.html. Accessed: May 2022.

[80] Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting models from source code in software modernization. *Software and Systems Modeling*, 13(2):713–734, 2014. doi:10.1007/s10270-012-0270-z.

[81] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. doi:10.1016/j.scico.2007.08.002.

[82] JSON Schema. http://json-schema.org/. Accessed: May 2022.

[83] David Kensche, Christoph Quix, Mohamed Amine Chatti, and Matthias Jarke. Gerome: A generic role based metamodel for model management. *Journal on Data Semantics*, 8:82–117, 2007. doi:10.1007/978-3-540-70664-9_4.

[84] Meike Klettke, Uta Störl, and Stefanie Scherzinger. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Conference on Database Systems for Business, Technology, and Web (BTW)*, pages 425–444, 2015.

[85] Dimitrios S. Kolovos, Fady Medhat, Richard F. Paige, Davide Di Ruscio, Tijs van der Storm, Sebastian Scholze, and Athanasios Zolotas. Domain-specific languages for the design, deployment and manipulation of heterogeneous databases. In *Proceedings of the 11th International Workshop on Modelling in Software Engineerings, MiSE@ICSE 2019, Montreal, QC, Canada, May, 2019*, pages 89–92. ACM. doi:10.1109/MiSE.2019.00021.

[86] Loup Meurice and Anthony Cleve. Supporting schema evolution in schema-less nosql data stores. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 457–461. IEEE Computer Society, 2017. doi:10.1109/SANER.2017.7884653.

[87] Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of dynamic database usage in java systems. In *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June, 2016*, volume 9694, pages 491–506. Springer. doi:10.1007/978-3-319-39696-5_30.

[88] Loup Meurice, Francisco Javier Bermudez Ruiz, Jens H. Weber, and Anthony Cleve. Establishing referential integrity in legacy information systems - reality bites! In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September, 2014*, pages 461–465. IEEE Computer Society. doi:10.1109/ICSME.2014.74.

[89] Michael J. Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu. NoSE: Schema Design for NoSQL Applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(10):2275–2289, 2017. doi:10.1109/TKDE.2017.2722412.

[90] Eclipse Foundation: Java MoDisco Metamodel. https://help.eclipse.org/latest/topic/org.eclipse.modisco.java.doc/mediawiki/java_metamodel/user.html. Accessed: May 2022.

[91] Eclipse Foundation: C Sharp MoDisco Metamodel. https://wiki.eclipse.org/MoDisco/CSharp. Accessed: May 2022.

[92] Chuang-Hue Moh, Ee-Peng Lim, and Wee Keong Ng. 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, 2000*, pages 144–151. IEEE Computer Society, 2000. doi:10.1109/WECWIS.2000.853869.

[93] Pablo D. Muñoz-Sánchez, Carlos Javier Fernández Candel, Jesús García Molina, and Diego Sevilla Ruiz. Managing physical schemas in mongodb stores. In *Advances in Conceptual Modeling - ER 2020 Workshops, CoMoNoS, Vienna, Austria, November, 2020*, volume 12584, pages 162–172. Springer. doi:10.1007/978-3-030-65847-2_15.

[94] Csaba Nagy and Anthony Cleve. Sqlinspect: a static analyzer to inspect database usage in java applications. In *Proceedings of the 40th International Conference on Software*

Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May, 2018, pages 93–96. ACM. doi:10.1145/3183440.3183496.

[95] Neo4j Webpage. Accessed: May 2022. URL: https://neo4j.com/.

[96] OrientDB Community Webpage. Accessed: May 2022. URL: http://orientdb.com/orientdb/.

[97] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. In *Journal of Management Information Systems*, volume 24, pages 45–77, September 2008. doi:10.2753/MIS0742-1222240302.

[98] Sadalage Pramod and Martin Fowler. *NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[99] Sudha Ram. Heterogeneous distributed database systems - guest editor's introduction. *Computer*, 24(12):7–10, 1991. doi:10.1109/MC.1991.10118.

[100] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with objects - the OOram software engineering method*. Manning, 1996.

[101] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. Inferring versioned schemas from nosql databases and its applications. In *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October, 2015*, volume 9381, pages 467–480. Springer. doi:10.1007/978-3-319-25264-3_35.

[102] Francisco Javier Bermudez Ruiz, Jesús García Molina, and Oscar Díaz García. On the application of model-driven engineering in data reengineering. *Information Systems*, 72:136–160, 2017. doi:10.1016/j.is.2017.10.004.

[103] Francisco Javier Bermudez Ruiz, Jesús García Molina, and Oscar Díaz García. On the application of model-driven engineering in data reengineering. *Information Systems*, 72:136–160, 2017. doi:10.1016/j.is.2017.10.004.

[104] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manual*. Addison-Wesley-Longman, 1999.

[105] Jennifer Ruttan. *The Architecture of Open Source Applications, Volume II: Structure, Scale, and a Few More Fearless Hacks*. Lulu.com, 2008.

[106] Herbert A. Simon. Prospects for cognitive science. In *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1988, Tokyo, Japan, November 28-December 2, 1988*, pages 111–119. OHMSHA Ltd. Tokyo and Springer-Verlag, 1988.

[107] Sirius Official Website, 2017. Accessed: May 2022. URL: https://eclipse.org/sirius/.

[108] Yannis Smaragdakis, Christoph Csallner, and Ranjith Subramanian. Scalable Satisfiability Checking and Test Data Generation From Modeling Diagrams. *Automated Software Engineering*, 16(1):73, 2009. doi:10.1007/s10515-008-0044-6.

[109] Apache spark webpage. Accessed: May 2022. URL: https://spark.apache.org/.

[110] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[111] Michael Stonebraker. The case for polystores. *ACM Sigmod Blog*, 2015. URL: https://wp.sigmod.org/?p=1629.

[112] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings International Symposium on Principles of Software Evolution*, pages 154–164. IEEE, 2000. doi:10.1109/ISPSE.2000.913233.

[113] Dionysios C. Tsichritzis and Frederick H. Lochovsky. *Data Models*. Prentice Hall Professional Technical Reference, 1982.

[114] Typhon Project. Hybrid Polystore Modelling Language (Final Version). Technical report, University of L'Aquila, 2018. URL: https://4d97e142-6f1b-4bbd-9bbb-577958797a89.filesusr.com/ugd/d3bb5c_3394b40f9cb54bcbb873f2c4ea1f2298.pdf.

[115] Vijay Vaishnavi and Bill Kuechler. Design Science Research in Information Systems. http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf, 2015. Accessed: May 2022.

[116] Variety repository. Accessed: May 2022. URL: https://github.com/variety/variety.

[117] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. URL: http://www.dslbook.org.

[118] Allen Wang. Unified Data Modeling for Relational and NoSQL Databases. InfoQ: Software Development News, Trends & Best Practices. https://www.infoq.com/articles/unified-data-modeling-for-relational-and-nosql-databases/, 2016.

[119] Lanjun Wang, Oktie Hassanzadeh, Shuo Zhang, Juwei Shi, Limei Jiao, Jia Zou, and Chen Wang. Schema management for document stores. *Proceedings VLDB Endow.*, 8(9):922–933, 2015. doi:10.14778/2777598.2777601.

[120] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014. doi:10.1007/978-3-662-43839-8.