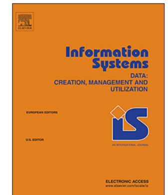
Information Systems 104 (2022) 101898

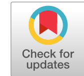
Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



A unified metamodel for NoSQL and relational databases ☆



Carlos J. Fernández Candel, Diego Sevilla Ruiz, Jesús J. García-Molina *

Faculty of Computer Science, University of Murcia, Murcia, Spain

A R T I C L E   I N F O

Article history:
Received 19 July 2021
Received in revised form 10 September 2021
Accepted 11 September 2021
Available online 25 September 2021
Recommended by Dennis Shasha

Keywords:
Unified metamodel
NoSQL databases
Schemaless
Schema inference
Model-driven engineering

A B S T R A C T

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.

Multi-model database tools normally use a generic or unified metamodel to represent schemas of the data model 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, and integrability is favored.

In this paper, 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) as well as 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 will be discussed, and some applications of U-Schema will be shown.

To achieve flexibility to respond to data changes, most of NoSQL systems are "schema-on-read," and the declaration of schemas is not required. 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. Through the paper, we will show how all these issues have been tackled in our approach.

As far as we know, no proposal exists in the literature of a unified metamodel for relational and the NoSQL paradigms which describes how each individual data model is integrated and mapped. Our metamodel goes beyond the existing proposals by distinguishing entity types and relationship types, representing aggregation and reference relationships, and including the notion of structural variability. Our contributions also include developing schema extraction strategies for schemaless systems of each NoSQL data model, and tackling performance and scalability in the implementation for each store.

© 2021 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

With the advent of modern data-intensive applications (e.g., Big Data, social networks, or IoT), NoSQL (*N*ot 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 classified in several database paradigms, but commonly the term NoSQL refers to four of them: *columnar*, *key–value*, *document*, and *graph* systems. 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. Therefore, we will assume here the data model of the most popular stores of each category. Table 1 shows the main features of the four mentioned NoSQL paradigms.

Over last years, as the popularity of NoSQL systems increased,[1] *polyglot persistence* (a new term coined for heterogeneous database systems) has been gaining acceptance as the data architecture of the future: applications using the set of databases that better fit their needs. Today, relational databases are still clearly the most used by a wide margin, but most popular relational

☆ This work has been funded by the Spanish Ministry of Science and Innovation (project grant TIN2017-86853-P).

* Corresponding author.
E-mail addresses: cjferna@um.es (C.J.F. Candel), dsevilla@um.es (D. Sevilla Ruiz), jmolina@um.es (J.J. García-Molina).

[1] Four of the top 10 being NoSQL systems in the DB engines ranking (https://db-engines.com/en/ranking) as of January 2021: MongoDB (5th), Redis (7th), Elasticsearch (8th), and Cassandra (10th).

https://doi.org/10.1016/j.is.2021.101898
0306-4379/© 2021 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Table 1
Types of NoSQL systems and some example implementations.

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

systems are evolving to support NoSQL features.[2] Two facts have motivated this interest in *polyglot persistence* [1,2]: (i) the complexity and variety of data to be managed by software systems, and (ii) a single type of database system does not fit all the needs of an increasing number of systems (e.g., learning management systems, online retail systems, or social networks.)

The successful adoption of NoSQL requires database tools similar to those available for relational systems. This entails to investigate how common database utilities can be available for NoSQL systems. In addition, these 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. In the case of data modeling tools, the shift towards multi-model solutions is evident: the most popular relational modeling tools are being extended to integrate NoSQL stores (e.g. ErWin[3] and ER/Studio[4] provide functionality for MongoDB) and new tools supporting a number of relational and NoSQL databases have appeared (e.g. Hackolade[5]). This multi-model nature should be considered for database tooling, i.e., tools should support multiple data models.

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. Being *schemaless* does not mean the absence of a schema, but that the schema information is implicit in data and code. Therefore, the schemas implicit in NoSQL stores must be reverse engineered in order to build a cohesive set of utilities for NoSQL databases such as schema viewers, query optimizers, or code generators. 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*. Recently, several NoSQL schema extraction approaches have been published [3–5], and some data modeling tools such as those mentioned above provide some kind of reverse engineering functionality.

When building multi-model database tools, the definition of a generic, universal, or unified metamodel can provide some benefits [6–10]. 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-independent. With the predominance of relational databases, the interest in multi-model tools declined, and little attention has been paid to the definition of unified metamodels. A remarkable proposal is the DB-Main approach [6,11] that defined the GER generic metamodel based on the EER (Extended Entity Relationship) data model. More recently, some universal metamodels [8, 9] have been created in the context of "Model Management" [7, 12]. With the emergence of NoSQL systems, some unified metamodels have been proposed to have a uniform access to data [8, 13], and the idea of an unified metamodel for data modeling tools was outlined in [10].

Unified data models are also used to define generic query language. Three decades ago, the ODMG standard [14] specifies a unified object-oriented data model and a SQL-like generic query language to make it easier for programmers to use object-oriented databases. Currently, the widespread use of NoSQL and other kinds of data stores along with relational systems, is motivating the definition of some generic data models on which generic query languages are defined, for example, Amazon has developed the PartiQL data model and query language [15] 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.

In the past years, we defined a reverse engineering strategy to infer logical schemas from document NoSQL databases [3], and have presented approaches to visualize inferred schemas [16] and automatically generate object-document mappers code [17] from schemas. Currently, we are tackling the definition of a synthetic data generation approach [18], and the extraction of implicit physical schemas from NoSQL data [19]. As our intention is to build multi-model database utilities, we have tackled the definition of a unified metamodel named U-Schema, which is capable of representing schemas for the four most common NoSQL data models, as well as the relational model. In this paper, we present U-Schema jointly with a data model for each NoSQL paradigm, and the mapping from those data models to U-Schema (*forward mappings*), and from U-Schema to the data models (*reverse mappings*). Common strategies are defined to implement and validate the mappings. Several applications of U-Schema are then commented.

The **research contributions** of this work are as follows:

1. To our knowledge, we present the first unified metamodel able to represent logical schemas both for the four most common kinds of NoSQL systems and relational systems. Two salient features of our proposal are: (i) U-Schema includes the notion of *structural variation* for entity and relationship types, as most NoSQL systems are schemaless, and (ii) unlike other proposals, the four kinds of relationships between entities that are typical in logical data modeling are supported by U-Schema: aggregation, references, graph relationships, and generalization. Capturing

[2] Top-8 systems in the DB engines ranking (https://db-engines.com/en/ranking are multi-model.

[3] http://erwin.com/products/erwin-data-modeler.

[4] https://www.idera.com/er-studio-enterprise-architecture-solutions.

[5] https://hackolade.com/.

structural variability allows us to accurately describe the structure of the stored data.

2. Defining the forward and reverse mappings between U-Schema and the data models that it integrates, we have established the notion of *canonical mapping* in which there is a natural correspondence between each element of a data model and elements of U-Schema. For each data model, its canonical mapping has been formally defined, as well as the reverse mapping for characteristics not present in the considered data model (e.g. graph systems do not support aggregate relationships, or structural variation is not possible in relational tables.)
3. A common strategy is proposed to extract unified schemas from databases. This strategy has been applied to implement canonical mappings for each paradigm integrated in U-Schema. As far as we know, all the schema extraction proposals for NoSQL stores, only consider one kind of store, normally document-based [4,5] or graph [20]. Our strategy also takes into account scalability and performance, using MapReduce processing on the native data.
4. We provide insights on how U-Schema can ease the implementation of database utilities in multi-model and multi-database environments such as database migration, schema queries, data generation for testing, query optimization, and schema visualization.
5. We show how U-Schema can act as generic metamodel to create a universal query language for relational and NoSQL stores.
6. We have defined the U-Schema metamodel with the Ecore metamodel, which is the central element of the Eclipse Modeling Framework (EMF) [21], a widely used open-source platform to develop *Model-Driven Software Engineering* (MDE) solutions [22]. Thus, the proposal is open to be used and incorporated in any future database tool development.

The rest of this paper is organized as follows. Next section will present the U-Schema data model. Then, Section 3 will describe the common process devised to reverse engineer implicit schemas from data, introduce a running example database, and will explain the common strategy applied to validate and assert the performance of the schema extraction algorithms. Next five sections will be devoted to define a logical data model, formally specify the mapping between U-Schema and the data model defined, and show the implementation and validation of the corresponding forward mapping. Once presented the unified metamodel and the mappings, we will discuss how they can be applied in common database tasks. Finally, we will contrast our proposal with related works, draw some conclusions, and outline further work.

2. The U-Schema unified data model

2.1. Logical modeling concepts in U-Schema

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.)

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) [23] and *UML Class Models* [24]: 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 [25]), 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 $P^{\varepsilon} = \{P_i^{\varepsilon}\}, i = 1 \ldots n$ 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 2 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 4 to 7 .

2.2. 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. Fig. 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`). 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. `StructuralFeatures` determine the structure of database objects, and include `Attributes` and `Aggregates`, while logical features specify what identifies an object (`Key`), and which `References` an object has to other objects.

Each attribute has a name and a data type. The data types included are: `Primitive` (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 Fig. 1, `StructuralVariations` 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) [21], a framework widely used to develop Model-Driven Software Engineering (MDE) solutions [22]. 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 [12,26]. 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 [27].

2.3. 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 [4,5], 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.

Table 2
Mapping between logical modeling concepts and NoSQL/relational database systems.

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

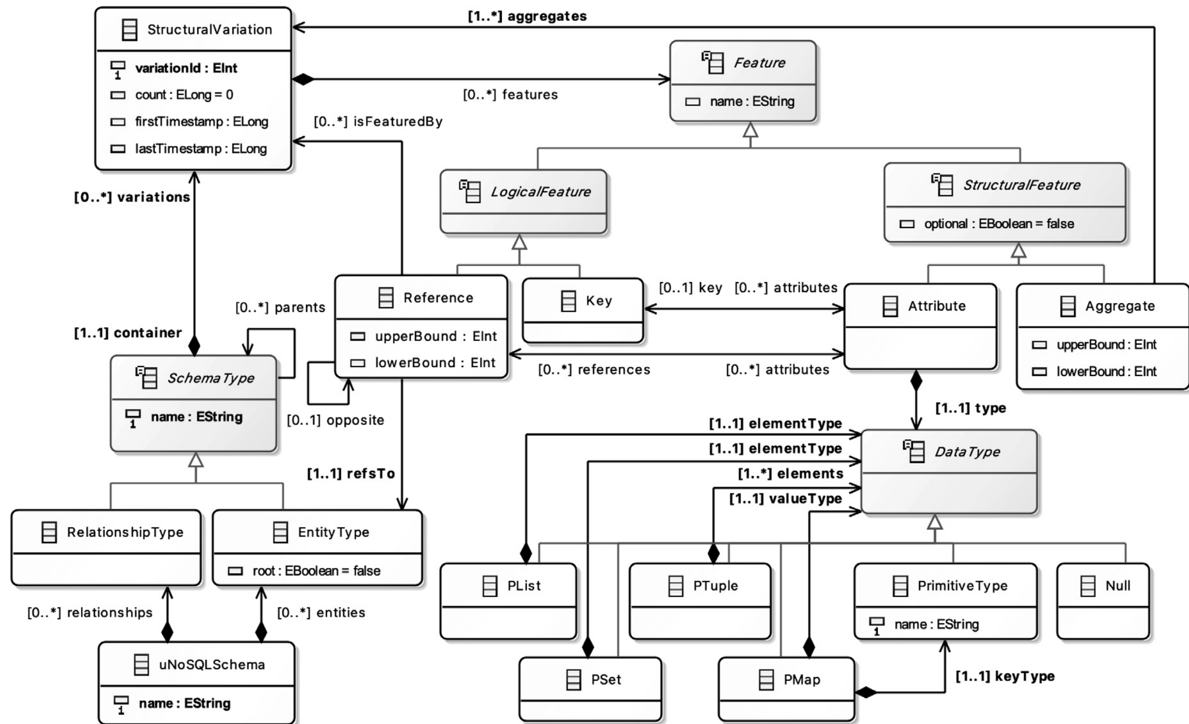


Fig. 1. U-Schema metamodel.

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 3.2 is shown in Fig. 9(b). `StructuralVariations` 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.

2.4. 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 3, 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 Section, the common strategies devised to implement and validate all the forward mappings will be described. In Sections 4 to 8 , 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:

  $u \leftrightarrow m \parallel$ {list of *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.

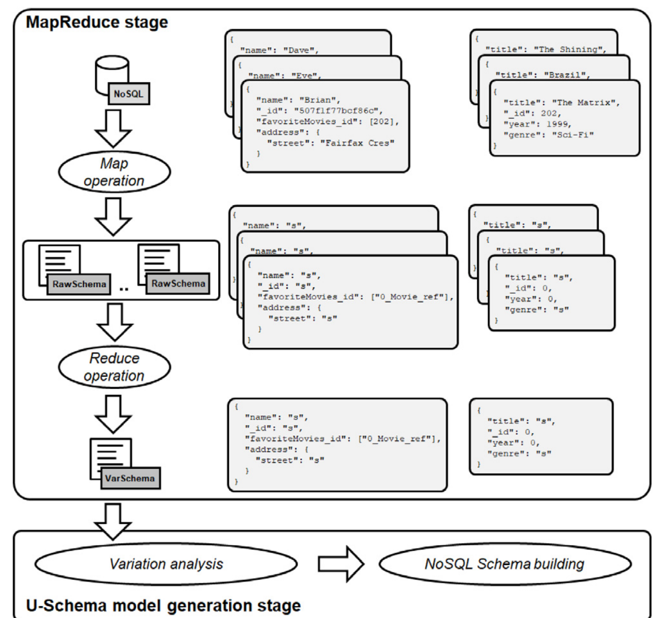


Fig. 2. Generic schema extraction strategy.

- 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.

3. A common strategy for the implementation and validation of the extraction of U-Schema models

In this Section, we will first explain how U-Schema models are built 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.

3.1. Building U-Schema models

As indicated in Section 2.4, 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 Fig. 2. 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.

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.

Fig. 2 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*. Fig. 2 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 4.4 and 6.4.

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.

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 [28]. 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.

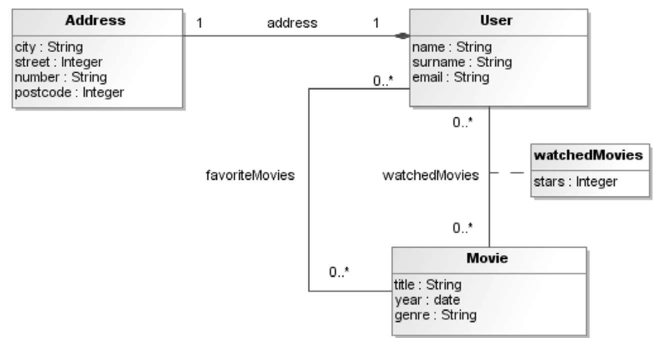


Fig. 3. "User profile" running example schema.

3.2. The "User Profiles" running example

Fig. 3 shows a "User Profiles" conceptual schema that will be used to build a database example for each paradigm integrated in our unified model. In each case, a "User Profiles" database will be populated to execute the algorithm that creates the corresponding U-Schema model and also to validate this algorithm as explained in the next subsection.

"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.

3.3. 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, and 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

Table 3
Database sizes.

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

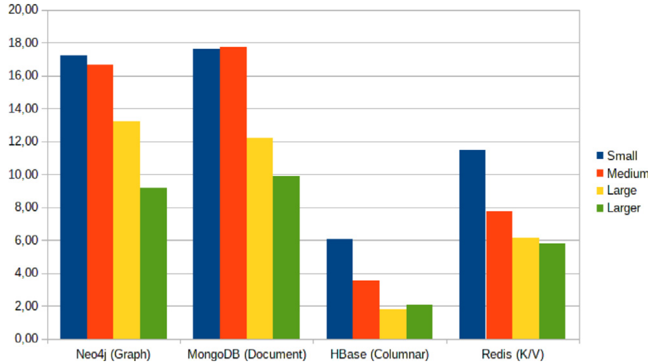


Fig. 4. Inference to query time ratio.

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 3.

All the performance tests were run on an Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz 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 4 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 3. 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 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. In the following sections, these results will be studied.

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

Table 4
Times for inference and queries for all the database implementations.

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

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.

4. Representing graph databases as U-Schema models

4.1. A data model for graph databases

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* [25], 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 Section 2.1.

For this kind of graph store, we have abstracted the following notion of logical *graph data model*, which is represented in form of UML class diagram in Fig. 5:

(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.

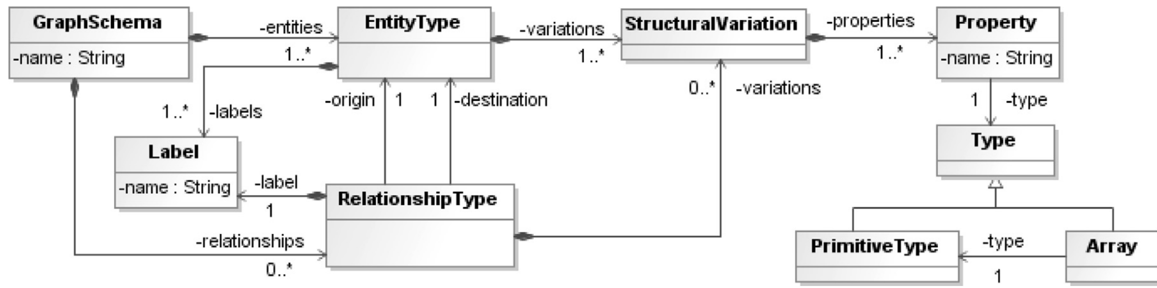


Fig. 5. Graph data model.

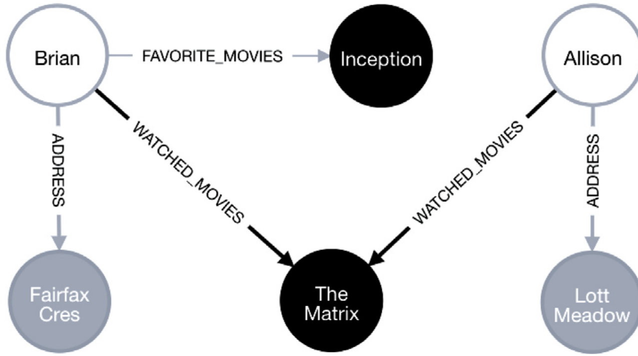


Fig. 6. "User Profiles" graph database example.

(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.

Table 2 shows the correspondence between graph database elements and the graph model elements expressed above. Note that a graph schema is obtained by the MapReduce operation of the schema extraction process described in the previous section.

Fig. 6 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 3.2.

4.2. Canonical mapping between graph model and U-Schema

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:

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

`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 `StructuralVariation` 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 `StructuralVariation` 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:

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

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.

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

Fig. 7. Example person data with address aggregate.

```
CREATE
  (:Person {name: "Diego"})
    -[:AGGR_address_address1 {}] ->
  (:Address {street: "Espinardo Campus", number: 2})
```

Fig. 8. Person aggregates address in Neo4j.

(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.

4.3. 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*). Fig. 7 shows an example JSON document of an U-Schema entity type `Person` that aggregates an object of the entity type `Address`. Fig. 8 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.)

4.4. Implementation and validation of the forward mapping for Neo4j

A slightly revised strategy to that described in Section 3 has been applied to implement the forward canonical mapping for Neo4j. We chose this store because it is the most popular graph database.[6] 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 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 [29]. 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 Fig. 9(a). We also show the *union schema* in Fig. 9(b).

The two experiments introduced in Section 3.3 were successfully carried out on the Neo4j database created for the running example and a *Movies* dataset available at the Neo4j website.[7]

Regarding scalability and efficiency of the model creation process, Table 4 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. Representing document databasesmas U-Schema models

5.1. A data model for document databases

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 [30,31] 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 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 [31]. 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, we have abstracted the following notion of *document data model*, which is represented in form of a UML class diagram in Fig. 10:

(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.

6 DB-Engines Ranking https://db-engines.com/en/ranking, (January, 2021).

7 No longer available at the original site, a copy can be obtained from https://github.com/catedrasaes-umu/NoSQLDataEngineering/blob/master/data/Neo4j/Movies/.

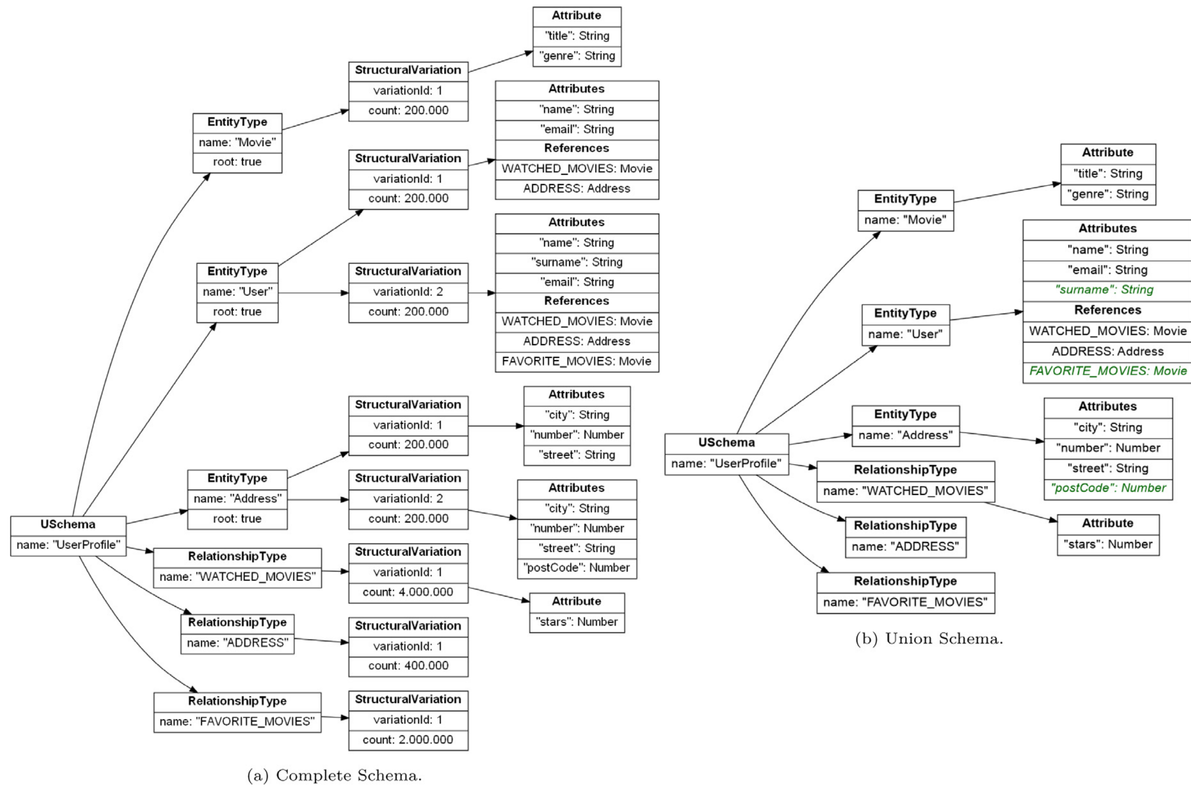


Fig. 9. "User Profiles" schema for graph stores.

(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.

Table 2 shows the correspondence between document database elements and the document model elements expressed above. Note that a document schema is obtained by the MapReduce operation of the schema extraction process described in Section 3.

Fig. 11 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.

5.2. 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:

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

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

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

$uS.entities$ 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.

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

`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.

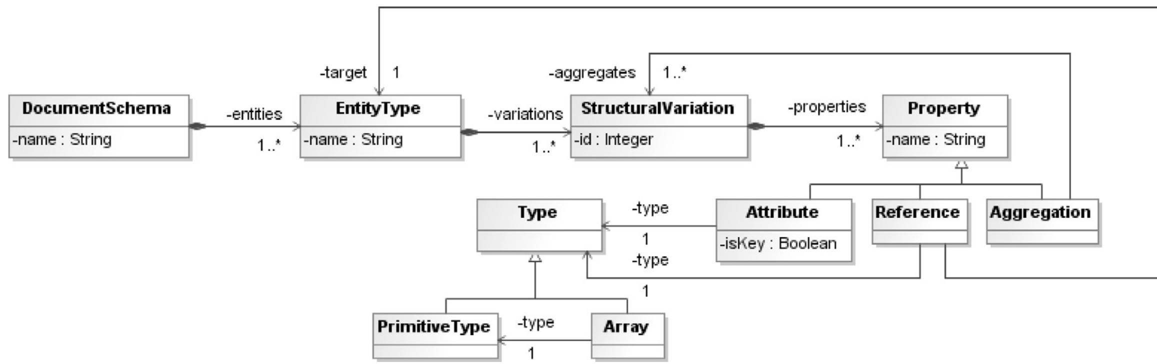


Fig. 10. Document data model.

User Collection

Property Name	Property Value
id	156
name	"Allison"
email	"allison@gmail.com"
watchedMovies (Array of Aggregated objects)	[0] stars: 3; movie_id: 202 (REF) / [1] stars: 5; movie_id: 295 (REF)
address (Aggregated object)	*city*: "Aylesbury"; *street*: "Lott Meadow"; *number*: 8

Property Name	Property Value
id	178
name	"Brian"
surname	"Caldwell" (SPECIFIC)
email	"brian_caldwell@gmail.com"
watchedMovies (Array of aggregated objects)	[0] stars: 4; movie_id: 202 (REF)
address (Aggregated object)	*city*: "Aylesbury"; *street*: "Fairfax Cres"; *number*: 6; *postcode*: 30760 (SPECIFIC)
favoriteMovies	[202, 267, 378] (SPECIFIC, REFS)

Movies Collection

Property Name	Property Value
id	202
title	"The Matrix"
year	1999
genre	"Science Fiction"

Fig. 11. "User Profiles" document database example.

(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. Reverse mapping completeness

The only element of U-Schema that is not directly supported by the document model is the `RelationshipType`. `RelationshipTypes` have structural variations, and some `References` 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.

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

Fig. 12 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 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 Fig. 10 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.

5.4. Implementation and validation of the forward mapping for MongoDB

Once the output of the MapReduce described in Section 3.1 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 `EntityTypes` and match the properties to identify the `StructuralVariations`.

The schema that would be inferred for the running example is shown in Fig. 13(a) and the union schema in Fig. 13(b).

The common validation strategy of Section 3.3 was successfully applied in MongoDB, with a database created for the running example, and with the *EveryPolitician* dataset.[8]

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

As with Neo4j, MongoDB shows worse ratio cases than with other two database implementations, as shown in Table 4. 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.

6. Representing key–value databases as U-Schema models

6.1. A data model for key–value databases

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[9] to define the canonical mapping between K/V databases and U-Schema, to which we will call the *flattened key pattern of compound objects* or simply *flattened object-key pattern*.

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 [32]. 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 in turn can have properties (e.g., "`<id>:<property>[<index>].<aggregated-property>`"). Fig. 14 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.

K/V systems are schemaless, and several structural variations of an entity type can therefore exist in the database. In Fig. 14, 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 5.1, and shown in Fig. 10, 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. This data model, as the Document model, has all the elements of U-Schema except the `RelationshipType` element, as shown in Table 2. We will use the term *aggregate-oriented*

[9] https://redislabs.com/redis-best-practices/data-storage-patterns/object-hash-storage/.

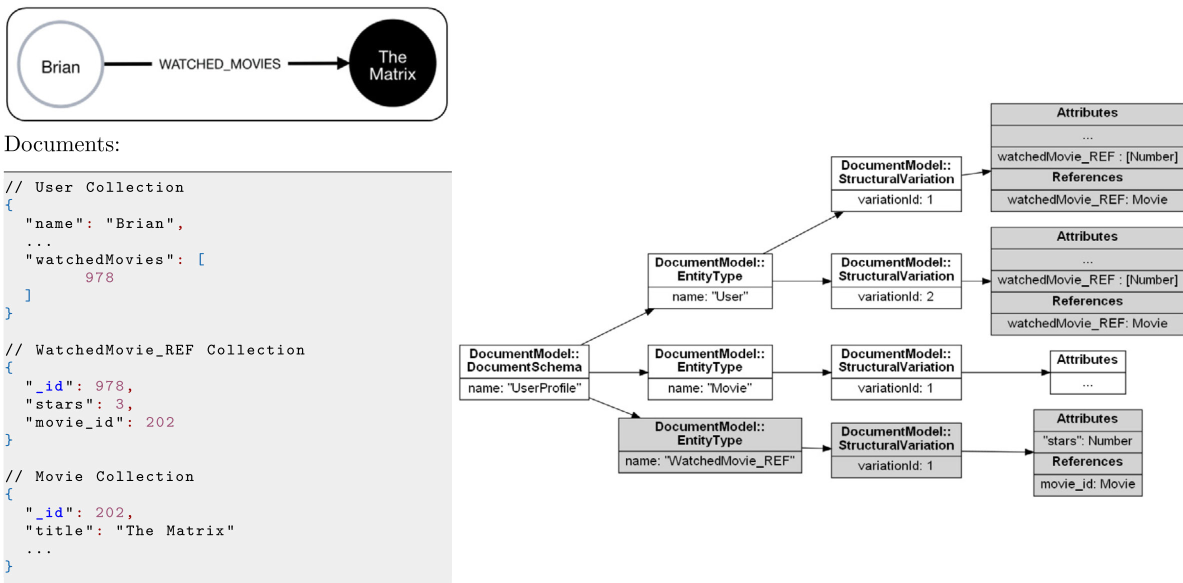


Fig. 12. Example of application of the reverse mapping from a `RelationshipType` of U-Schema to a document schema.

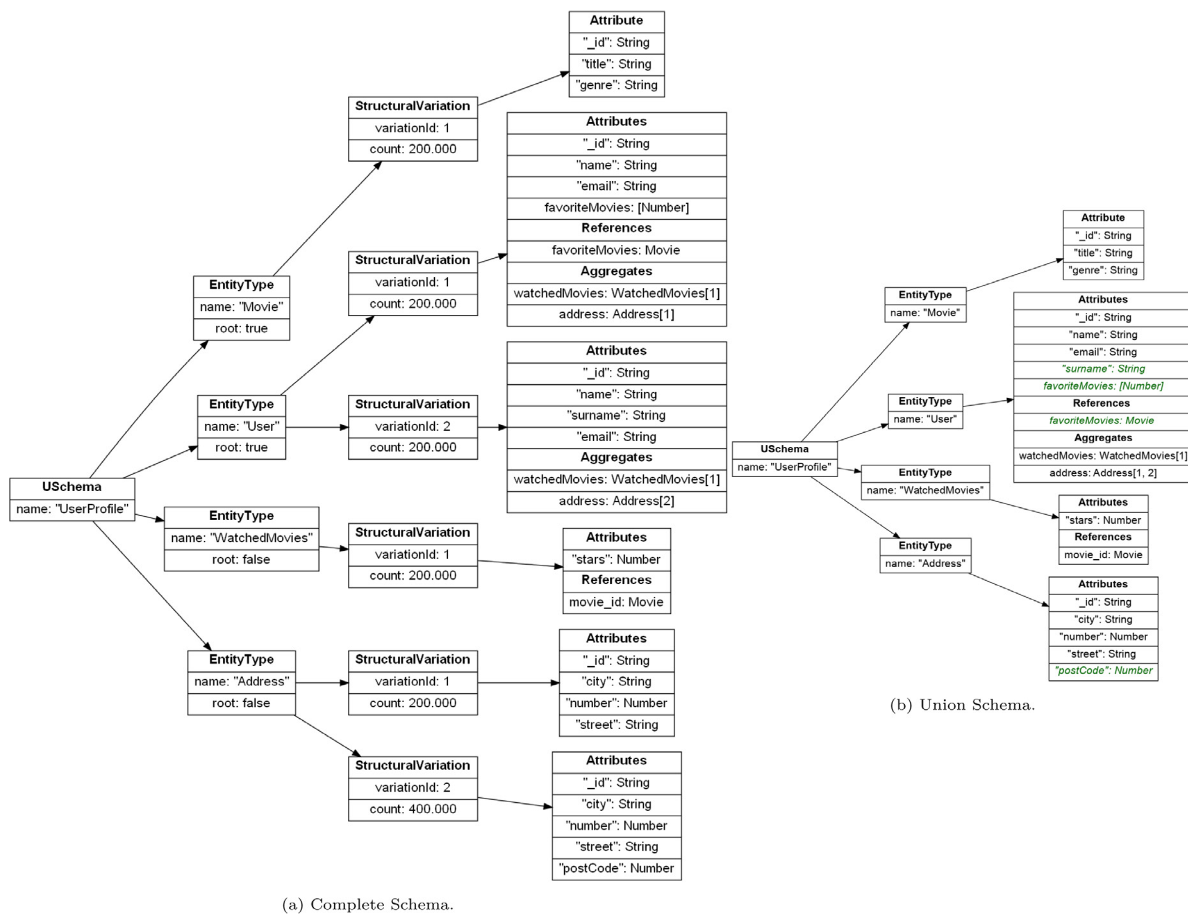


Fig. 13. "User Profiles" schema and union schema for document stores.

data model to group the Document, Key–Value, and Columnar data models, as suggested in [2], 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.

	Key	Value
1	**User**:156:*name*	"Allison"
2	**User**:156:*email*	"allison@gmail.com"
3	**User**:156:*watchedMovies[0].stars*	3
4	**User**:156:*watchedMovies[0].movie_id*	202 REF
5	**User**:156:*watchedMovies[1].stars*	5
6	**User**:156:*watchedMovies[1].movie_id*	295 REF
7	**User**:156:*address.city*	"Aylesbury"
8	**User**:156:*address.street*	"Lott Meadow"
9	**User**:156:*address.number*	8
10	**User**:178:*name*	"Brian"
11	**User**:178:*surname*	"Caldwell" SPECIFIC
12	**User**:178:*email*	"brian_caldwen@gmail.com"
13	**User**:178:*watchedMovies[0].stars*	4
14	**User**:178:*watchedMovies[0].movie_id*	202 REF
15	**User**:178:*address.city*	"Aylesbury"
16	**User**:178:*address.street*	"Fairfax Cres"
17	**User**:178:*address.number*	6
18	**User**:178:*address.postcode*	30760 SPECIFIC
19	**User**:178:*favoriteMovies*	[202, 267, 378] SPECIFIC REFS
20	**Movie**:202:*title*	"The Matrix"
21	**Movie**:202:*year*	1999
22	**Movie**:202:*genre*	"Science Fiction"

Fig. 14. Key–value database example for running example.

Regarding the data types of values, they usually provide basic scalar types as well as common collection types.

6.2. Canonical mapping between key–value model and U-Schema

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`.

6.3. Reverse mapping completeness

The same reverse completeness mapping rules exposed in Section 5.3 for the document model are applicable in this case.

6.4. 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.[10]

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 [29] 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 Fig. 13(a). The union schema is shown in Fig. 13(b).

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 5.4) 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.

7. Representing columnar databases as U-Schema models

7.1. A data model for columnar databases

In columnar databases, data is structured in a similar way to relational databases. In the most popular columnar databases (Hbase [33] and Cassandra [34]),[11] 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

[10] As shown in https://db-engines.com/en/ranking. Redis appears in the 7th position (March, 2021).

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

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*. Fig. 15 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 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.

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 Fig. 15, 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*[12] (“<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 Fig. 15.

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*.

7.2. Canonical mapping between columnar databases and 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.

7.3. 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. 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.

[12] https://hbase.apache.org/book.html#schema.casestudies.custorder.obj.denorm.

7.4. 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 3.1, 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 3.3. A database was created for the running example, and the same *EveryPolitician* real world dataset used in MongoDB and Redis, introduced in Section 5.4, was injected into Hbase and Cassandra.

Figs. 13(a) and 13(b) show the variation schemas obtained for the running example, which are the same for all the aggregation-based stores.

As shown in Table 4 and Fig. 4, 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.

8. Representing relational databases as U-Schema models

8.1. 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* [35]. A detailed description of the relational model can be found in [36,37].

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. Fig. 16 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.

User Table

Key: 156

User		Address			WatchedMovies			
name	email	city	street	number	0.stars	0.movie_id	1.stars	1.movie_id
"Allison"	"allison@gmail.com"	"Aylesbury"	"Lott Meadow"	8	3	202	5	295

Key: 178

User				Address				WatchedMovies	
name	surname	email	favoriteMovies	city	street	number	postcode	0.stars	0.movie_id
"Brian"	"Caldwell"	"brian_caldwell@gmail.com"	[202, 267, 378]	"Aylesbury"	"Fairfax Cres"	6	30760	4	202

Movie Table

Key: 202

Movie		
title	year	genre
"The Matrix"	1999	"Science Fiction"

Fig. 15. Columnar database example for the running example.

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

Fig. 16. "User Profile" relational example.

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) [36], Extended ER (EER) [37] and Object-Orientation modeling are the most widely used formalisms to model conceptual and logical schemas for relational databases. As explained in Section 2.1, the main concepts of such formalisms are included in U-Schema, which redefines most of them, and adds some other concepts.

8.2. Canonical mapping between relational model and 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):

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

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:

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

8.3. 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 `StructuralVariations`, 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 3.
- 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 Fig. 16.

8.4. 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, `Keys` are created for primary keys in tables, which will have references to `Attributes` that have been instantiated previously for the columns that are part of the primary key (R5). Finally, `References` 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.[13] 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 and scalability, as with Cassandra, relational databases have not been considered because schemas are already available.

9. Related work

In this section, the U-Schema metamodel will be contrasted to some relevant generic metamodels defined for database schemas, and the schema inference strategy to others published for NoSQL stores.

9.1. Generic metamodels

DB-Main was a long-term project aimed at tackling the problems related to database evolution [11,38]. 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 [11]. Here, our interest is focused on the two former elements. The generic GER metamodel was defined as an extension of the ER metamodel [36]. 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. Our proposal differs of the GER approach in several significant aspects.

A. Support of semi-structured data in NoSQL stores It is convenient to remark that our approach shares objectives with DB-Main. 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.

B. Physical and conceptual level separation in different metamodels U-Schema is intended to represent logical schemas, so that conceptual and physical schemas are separately modeled. Instead of mixing all the information in a single metamodel, we have considered more convenient to separate the large amount of physical concepts in their own metamodel and to have a simpler conceptual model. 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 paper. At this moment, we have defined a physical metamodel for MongoDB, as described in [19].

C. Concrete schemas are directly represented in U-Schema 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

[13] 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.

of rules that maps each data model to U-Schema determines the U-Schema elements involved, and therefore the valid structures.

D. Structural variation representation A central notion of U-Schema is structural variation. Variations of entity and relationship types can be represented. This information can be useful in different tasks. For example, variations are used to identify whether an entity type contains a type hierarchy (e.g. a *Product* hierarchy) in [39]. Variations also allow to analyze the database evolution, or can be used to generate test datasets, among other tasks.

E. Solution based on MDE specification As indicated in Section 2, we have defined U-Schema with the Ecore metamodeling language with the purpose of taking advantage of MDE technology integrated in the EMF framework [21].

F. Schema extraction 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.

Model Management (**MM**) is an approach aimed to solve *data programmability* problems which normally involve complex mappings between data schemas of different sources [7,12]. 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 [7], 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.

Two universal metamodels for applying Model Management are presented in [8] and [9]. **Paolo Atzeni et al.** [8] described a universal metamodel based on a three-level architecture similar to those defined in the EMF framework and used in our work: a metamodeling language (Ecore) is used to define metamodels (U-Schema in our case), which, in turn, are used to create models (schemas in our case). In [8], a set of 13 meta-constructs were defined to represent the concepts used in different data formalisms. For example, *Abstract* is proposed to model autonomous concepts such as ER entities or OO classes, *AbstractAttribute* to model references, and *Generalization* to model *Abstract* hierarchies. 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. The metamodel was accompanied by a basic tooling for textual and graphical visualization.

The main differences between this universal metamodel and U-Schema are the following:

A. A different purpose and meta-modeling architecture 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, as indicated in Section 3.1, 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.

B. Availability of tools for basic model operations While we 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 provide tools supporting model comparison (EMF Compare)[14] and model diff/merge operations (EMF Diff/Merge),[15] as well as model-transformation languages to implement the *modelgen* operator.

C. Relationship types and structural variations 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.

GeRoMe is another generic metamodel proposed for Model Management [9]. A *role-based modeling* is applied to define a metamodel able to represent different data models. In mid-nineties, role-based modeling approaches received attention in the context of object-oriented programming to model the multiple-classification and object collaborations [40]. However, that interest has decreased over the years because languages and tools do not support the notion of *role*. Extended ER, Relational, OWL-DL, XML Schema, and UML were analyzed in GeRoMe with the aim to identify their similarities and differences. Then, a set of roles was established, and the role-based metamodel created. 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 metamodels, and using a well-know metamodeling architecture.

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 DB-Main, the tool can currently be acquired from the Rever company[16] as a tool to simplify data engineering tasks.

More recently, several metamodels have been proposed to represent NoSQL schemas. **SOS** is a metamodel designed to represent schemas of aggregate-based stores [13]. With this uniform representation, 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. Later, SOS evolved to the **NoAM** (NoSQL Abstract Model) metamodel, which was defined as part of a design method for aggregate-based NoSQL databases [41,42]. NoAM was designed as an intermediate representation to transform aggregate objects of database applications into NoSQL data. A NoAM database 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. In [41], several strategies are described to represent a collection of aggregate objects in form of a NoAM database.

NoAM and SOS were 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.

14 https://www.eclipse.org/emf/compare/.
15 https://wiki.eclipse.org/EMF_DiffMerge.
16 https://www.dataengineers.eu/en/db-main/.

ERwin unified data modeler (ModelSet) is a project outlined in an article in infoQ [10] whose purpose is very close to ours. However, to our knowledge, results of that project have not been published yet. In [10], Allen Wang, responsible of the ERwin project, pointed out on the importance of "using logical models describing business requirements and de-normalizing schema to physical data models" in database design. A simple unified logical model is shown to represent three kind of schemas: columnar, document, and relational schema. The metamodel only includes four elements. The three basic modeling constructs: *Entity*, *Relationships*, *Properties* (of entities), and *Tags* are used to add additional information to basic constructs. A physical model should be built for each system. Query and data production patterns are defined on the logical model for its transformation into physical model. Several significant differences are found between U-Schema and the ERwin metamodel: (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.

The Typhon project[17] 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 [43] 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. Some remarkable differences with U-Schema are: (i) TyphonML is not a language defined on a unified metamodel, instead U-Schema is separated from any schema declaration language. In fact, we have created the Athena language on U-Schema, and other languages could be defined[18]; (ii) The existence of structural variation in NoSQL systems is not considered; (iii) As can be observed in [43], for each paradigm, the TyphonML metamodel includes logical and physical aspects; Instead, our choice is to separate both levels of abstraction in two metamodels as pointed out in [19]; (iv) Although graph stores are represented, the concept of relationship type is not included in TyphonML; (v) Aggregates are not represented 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 U-Schema represents aggregates and references with two metaclasses, 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.

A 2-step model transformation chain aimed to transform conceptual models into physical schemas (Cassandra stores are considered) is described in [44]. Logical models are generated in an intermediate step. Conceptual schemas and logical schemas are represented by means of very simple metamodels. A conceptual schema is formed by a set of classes and datatypes, and classes include attributes and relationships. A relational model is used as logical model, to which relationships are added. This proposal has some flaws such as (i) relationship types and references are not distinguished, which is necessary for graph schemas, (ii) references have not properties, and (iii) the separation between logical and conceptual model is not justified because they include the same concepts but different names; this can be observed in the very simple conceptual-to-logical transformation shown in the paper.

Table 5 summarizes the discussion made above, and compares the generics metamodels considered according to several criteria.

9.2. NoSQL schema extraction strategies

Recently, several approaches to extract schemas from NoSQL document stores have been published [4,5,45]. A detailed study of these works can be found in [46] where they were contrasted to our previous approach for document stores [3]. Moreover, some works on schema extraction from JSON datasets have also been presented. In [45], a MapReduce is used 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 in the dataset, and the value records the number of elements of the that type. In a second step, heuristics are applied to merge similar types.

The main differences of the approach described here with previous extraction strategies are the following: (i) They are focused on document stores. Instead, we have defined a general strategy applicable to the four main NoSQL paradigms and the relational model; (ii) Like [45], our approach use a MapReduce operation to improve the efficiency of the inference process; (iii) Our inference strategy discovers relationships between entities, and structural variations of entities. (iv) 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 2.2.

An MDE-based reverse engineering approach for extracting conceptual graph schemas is described in [20]. `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. Our inference process differs of this strategy in several significant aspects, apart from being a generic strategy and use a MapReduce to considerably improve the efficiency: (i) We access stored data instead of using `CREATE` statements, which might not be available. (ii) Instead of building a graph model, we create a table with all relationships as input to the MapReduce operation; this table allows us to obtain the cardinality of each relationship. (iii) We obtain a logical schema that include structural variations for relationships and entities.

9.3. 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 relevant of these tools, we have found no evidence showing the use of a unified metamodel. Next, we contrast features of these 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 [47] and ERwin [48] 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.

17 https://www.typhon-project.org/.

18 https://catedrasaes-umu.github.io/NoSQLDataEngineering/tools.html.

Table 5
Approaches defining a generic metamodel.

	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

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 [49] 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 [50] 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.

10. 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.

10.1. 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 [15,51] 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 [52]. Both languages are based on the SQL standard due to its widespread adoption.

10.1.1. 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[19] 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. Recently, we published the results obtained in applying U-Schema to define an approach for NoSQL schema evolution [53, 54]. As part of that work, we defined the Athena [54] and Orion [53] languages to declare and evolve, respectively, platform-independent schemas.[20]

The U-Schema query engine would be implemented with a strategy similar to that applied for PartiQL [15,51], as illustrated in Fig. 17.

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 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.

[19] https://www.gqlstandards.org/.

[20] With "platform" we refer to data models and data stores.

10.1.2. 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 5.1: 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.

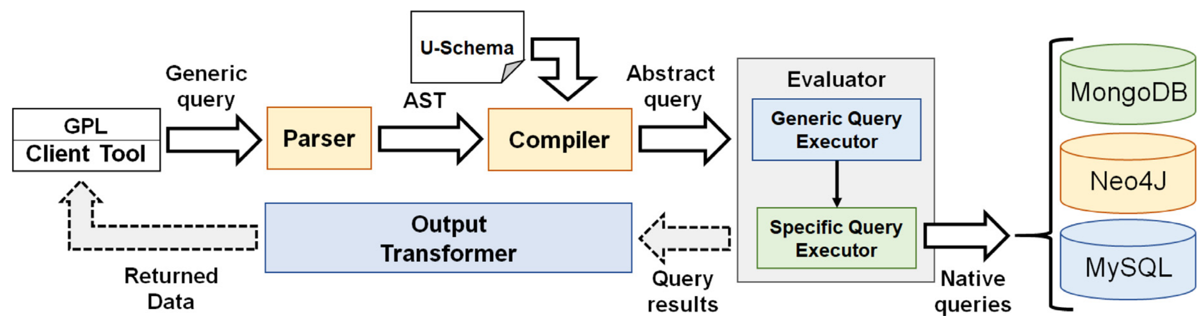


Fig. 17. Overview of a generic query architecture based on U-Schema.

```
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 4.1.

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 in the Deimos language [18] 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 [53].

10.2. 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 4 to 8 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 4.3. 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.

10.3. 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 [34] and OrientDB [55]. 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. [5]. 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. [5], which focused on MongoDB, and only suggested a couple of queries to illustrate the idea. A first version of our language can be downloaded.[21]

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).

10.4. 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 [56,57]. 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 [58]. 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.

We developed an initial version of a U-Schema-based data generation language with the described characteristics [18].

10.5. 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). In a previous work, we explored the visualization of document schemas and proposed several kinds of diagrams for document systems [16]. Now, 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.

11. Conclusions and future work

Multi-platform database engineering tools commonly define a unified metamodel to represent database schemas of a variety of systems. In this paper, we have presented a proposal of unified metamodel that integrates data models for relational and NoSQL systems (key–value, document, wide column, and graph). These systems cover most of current database applications. Our work is motivated by the growing interest in multimodel database tooling and systems as polyglot persistence is

[21] https://github.com/catedrasaes-umu/NoSQLDataEngineering.

considered essential to satisfy needs of modern applications. With U-Schema, we have defined a representation able to express inferred or declared schemas at a similar abstraction level to EER and Object-Orientation logical models.

U-Schema is the first logical unified metamodel defined for NoSQL and relational systems taking into account structural variation, relationship types, aggregations, and references. Through forward and reverse mappings, we have formally shown how U-Schema is able to represent each considered data model, and how U-Schema models can be converted to schemas of the individual models. We would like to remark that the extraction of schemas from databases (forward mappings) have been implemented for the most widely used NoSQL systems (Neo4j, MongoDB, Redis, Cassandra, and HBase), as well as for one of the most used relational systems (MySQL). For each extraction algorithm, scalability and performance were assessed. Having used the *de facto* standard Ecore to represent the schemas turns the framework in a reusable and adaptable tool, and Eclipse modeling tooling can be used to build database tooling.

Future work can be divided in two lines, depending on whether they have to do with the unified metamodeling approach, or with applications based on U-Schema. We approached the unified representation of schemas by separating logical and physical aspects. The metamodel presented here concerns to the logical view, and a new metamodel will represent the unified physical schemas. 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 [19]. 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. Finally, it is planned to build several tools and languages around U-Schema: (i) The migration approach outlined in Section 10.2, which will include a U-Schema based schema mapping language to express specialized mappings as described in Section 2.4; (ii) Complete the generic Schema Query Language introduced in Section 10.3; (iii) Define some diagramming tools from the results in [16]; (iv) A universal schema definition language, that, by using U-Schema, allows to define schemas homogeneously through NoSQL and relational datastores; (v) Support an approach of platform-independent schema evolution through the definition of a change taxonomy implemented by a schema change operation language; and (vi) This language could be used to explore the impact of schema changes on the set of queries of an application, following automated reinforcement learning techniques.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] M. Stonebraker, The case for polystores, ACM Sigmod Blog (2015) URL https://wp.sigmod.org/?p=1629.

[2] P. Sadalage, M. Fowler, NoSQL Distilled. A Brief Guide To the Emerging World of Polyglot Persistence, Addison-Wesley, 2012.

[3] D. Sevilla Ruiz, S. Feliciano Morales, J. García Molina, Inferring versioned schemas from NoSQL databases and its applications, in: 34th International Conference on Conceptual Modeling (ER), Stockholm, Sweden, 2015, pp. 467–480.

[4] M. Klettke, U. Störl, S. Scherzinger, Schema extraction and structural outlier detection for JSON-based NoSQL data stores, in: Conference on Database Systems for Business, Technology, and Web (BTW), 2015, pp. 425–444.

[5] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi, L. Jiao, J. Zou, C. Wang, Schema management for document stores, Proc. VLDB Endow. 8 (9) (2015) 922–933, http://dx.doi.org/10.14778/2777598.2777601.

[6] V. Englebert, J.-L. Hainaut, DB-Main: A next generation meta-case, Inf. Syst. 24 (2) (1999) 99–112.

[7] P.A. Bernstein, S. Melnik, Model management 2.0: manipulating richer mappings, in: Proceedings of the ACM SIGMOD Int. Conference on Management of Data, 2007, pp. 1–12.

[8] P. Atzeni, G. Gianforme, P. Cappellari, A universal metamodel and its dictionary, Trans. Large Scale Data Knowl. Centered Syst. 1 (2009) 38–62.

[9] D. Kensche, C. Quix, M.A. Chatti, M. Jarke, Gerome: A generic role based metamodel for model management, J. Data Semantics 8 (2007) 82–117.

[10] A. Wang, Unified data modeling for relational and NoSQL databases, Infoq (2016) URL https://www.infoq.com/articles/unified-data-modeling-for-relational-and-nosql-databases/.

[11] J.-M. Hick, J.-L. Hainaut, Strategy for database application evolution: The DB-MAIN approach, in: International Conference on Conceptual Modeling, Springer, 2003, pp. 291–306.

[12] P.A. Bernstein, A.Y. Halevy, R. Pottinger, A vision of management of complex models, SIGMOD Rec. 29 (4) (2000) 55–63.

[13] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to non-relational database systems: The SOS platform, in: 24th International Conference on Advanced Information Systems Engineering (CAiSE), Gdansk, Poland, 2012, pp. 160–174.

[14] R. Cattell, D.K. Barry, The Object Data Standard: ODMG 3.0, Morgan Kaufmann, 2000.

[15] PartiQL specification, 2021, URL https://partiql.org/assets/PartiQL-Specification.pdf (Accessed September 2021).

[16] A. Hernández Chillón, S. Feliciano Morales, D. Sevilla Ruiz, J. García Molina, Exploring the visualization of schemas for aggregate-oriented NoSQL databases, in: ER Forum 2017, 36th Int. Conf. on Conceptual Modeling (ER), Valencia, Spain, 2017, pp. 72–85.

[17] A. Hernández Chillón, D. Sevilla Ruiz, J. García Molina, S. Feliciano Morales, A model-driven approach to generate schemas for object-document mappers, IEEE Access 7 (2019) 59126–59142.

[18] A. Hernández Chillon, D. Sevilla Ruiz, J. Garcia-Molina, Deimos: A model-based NoSQL data generation language, in: CoMoNoS Workshop in Conceptual Modeling Int. Conf., 2020.

[19] P.M. noz, C.J. Fernández, J. Garcia-Molina, D. Sevilla Ruiz, Extracting physical and logical schemas for document stores, in: CoMoNoS Workshop in Conceptual Modeling Int. Conf., 2020.

[20] I. Comyn-Wattiau, J. Akoka, Model driven reverse engineering of NoSQL property graph databases: The case of Neo4j, in: 2017 IEEE International Conference on Big Data, 2017, Boston, MA, USA, December 11-14, 2017, 2017, pp. 453–458.

[21] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, Addison-Wesley Professional, 2009.

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

[23] P.P. shan Chen, The entity-relationship model: Toward a unified view of data, ACM Trans. Database Syst. 1 (1976) 9–36.

[24] J.E. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley-Longman, 1999.

[25] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, D. VrgoÄ, Foundations of modern query languages for graph databases, ACM Comput. Surv. 50 (5) (2017).

[26] J. 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, 2005, pp. 95–143, http://dx.doi.org/10.1007/11877028_4.

[27] F.J. Bermudez, J.G. Molina, O. Díaz, On the application of model-driven engineering in data reengineering, Inf. Syst. 72 (2017) 136–160.

[28] E. Gamma, R. Helm, R. Johnson, J.M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[29] Apache spark webpage, 2021, URL https://spark.apache.org/ (Accessed March 2021).

[30] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations To Semistructured Data and XML, Morgan Kaufmann, 1999.

[31] P. Buneman, Semistructured data, in: Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM, Arizona, USA, 1997, pp. 117–121.

[32] S. Gössner, JSONPath – XPath for JSON, 2020, URL https://tools.ietf.org/id/draft-goessner-dispatch-jsonpath-00.html.

[33] Hbase webpage, 2007, URL https://hbase.apache.org/ (Accessed September 2020).

[34] Cassandra webpage, 2016, URL http://cassandra.apache.org/ (Accessed September 2020).

[35] E. Codd, A relational model of data for large shared data banks, Commun. ACM 13 (6) (1970) 377–387, http://dx.doi.org/10.1145/362384.362685.

[36] D.C. Tsichritzis, F.H. Lochovsky, Data Models, Prentice Hall Professional Technical Reference, 1982.

[37] R. Elmasri, S.B. Navathe, Fundamentals of Database Systems, seventh ed., Pearson, 2015.
[38] J. Hainaut, V. Englebert, J. Henrard, J. Hick, D. Roland, Database evolution: the DB-main approach, in: P. Loucopoulos (Ed.), Entity-Relationship Approach - ER'94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, UK, December 13-16, 1994, Proceedings, in: Lecture Notes in Computer Science, vol. 881, Springer, 1994, pp. 112–131.
[39] A. Hernández Chillon, J.R. Hoyos, D. Sevilla Ruiz, J. Garcia-Molina, Discovering entity inheritance relationships in document stores, Knowl.-Based Syst. 230 (2021) 107394, http://dx.doi.org/10.1016/j.knosys.2021.107394.
[40] T. Reenskaug, P. Wold, O.A. Lehne, Working with Objects - The OOram Software Engineering Method, Manning, 1996.
[41] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to NoSQL systems, Inf. Syst. 43 (2014) 117–133, URL http://www.sciencedirect.com/science/article/pii/S0306437913000719.
[42] P. Atzeni, F. Bugiotti, L. Cabibbo, R. Torlone, Data modeling in the NoSQL world, Comput. Stand. Interfaces 67 (2020).
[43] T. 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.
[44] F. Abdelhédi, A.A. Brahim, F. Atigui, G. Zurfluh, Logical unified modeling for NoSQL databases, in: ICEIS (1), SciTePress, 2017, pp. 249–256.
[45] D. Colazzo, G. Ghelli, C. Sartiani, Typing massive JSON datasets, in: International Workshop on Cross-Model Language Design and Implementation, vol. 541, 2012, pp. 12–15.
[46] S. Feliciano., Inferring NoSQL data schemas with model-driven engineering techniques, (Ph.D. thesis), Faculty of Informatics. University of Murcia, Spain, 2017.
[47] ER-Studio webpage, 2015, URL https://www.idera.com/er-studio-enterprise-data-modeling-and-architecture-tools(Accessed February 2021).
[48] ERwin data modeler webpage, 2016, URL http://erwin.com/products/erwin-data-modeler (Accessed February 2021).
[49] Hackolade webpage, 2016, URL https://hackolade.com/ (Accessed September 2020).
[50] Dbschema webpage, 2016, URL http://www.dbschema.com (Accessed October 2018).
[51] Announcing PartiQL: One query language for all your data, 2021, URL https://aws.amazon.com/es/blogs/opensource/announcing-partiql-one-query-language-for-all-your-data/ (Accessed September 2021).
[52] OrientDB SQL reference, 2021, URL http://orientdb.com/docs/3.1.x/sql/ (Accessed September 2021).
[53] A. Hernández Chillón, D. Sevilla Ruiz, J. García Molina, Towards a taxonomy of schema changes for NoSQL databases: The orion language, in: ER 2021, 40th Int. Conf. on Conceptual Modeling (ER), St. John's, NL, Canada, 2021.
[54] A. Hernández Chillon, D. Sevilla Ruiz, J. Garcia-Molina, Athena: A database-independent schema definition language, in: CoMoNoS 2nd Workshop in Conceptual Modeling Int. Conf., 2021.
[55] OrientDB community webpage, 2020, URL http://orientdb.com/orientdb/ (Accessed September 2020).
[56] N. Bruno, S. Chaudhuri, Flexible database generators, in: 31st International Conference on VLDB, 2005, pp. 1097–1107.
[57] Y. Smaragdakis, et al., Scalable satisfiability checking and test data generation from modeling diagrams, Auto. Softw. Eng. 16 (1) (2009) 73.
[58] M.J. Mior, K. Salem, A. Aboulnaga, R. Liu, Nose: Schema design for NoSQL applications, IEEE Trans. Knowl. Data Eng. 29 (10) (2017) 2275–2289.