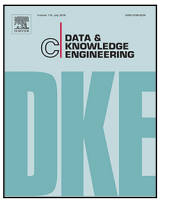
Data & Knowledge Engineering 145 (2023) 102149

Contents lists available at ScienceDirect

Data & Knowledge Engineering

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



Designing NoSQL databases based on multiple requirement views

Noa Roy-Hubara [a], Arnon Sturm [a,*], Peretz Shoval [a,b]

[a] Ben-Gurion University of the Negev, Beer Sheva, Israel
[b] Netanya Academic College, Netanya, Israel

ARTICLE INFO

Keywords:
NoSQL
Conceptual modeling
User requirements
Graph database
Document Database
Database design

ABSTRACT

In recent years, new data characteristics led to the development of new database management systems named NoSQL. As opposed to the mature Relational databases, design methods for the new databases receive little attention and mainly consider the data-related requirements. In this paper, we present methods for designing two types of NoSQL databases – Document and Graph databases – that consider not only the data-related but also functional-related requirements. We empirically evaluate the methods and apply the design methods to two leading database systems. We found that the databases designed with the consideration of functional requirements perform better, with respect to time of execution and database I/O operations, compared to when designed without considering them.

1. Introduction

The new digital era is characterized by multiple types of databases, including the "traditional" Relational model and the relatively new NoSQL databases, which comprise four subtypes: Document, Column-based, Key–value, and Graph databases. The proliferation of the new database systems required the creation of new database design methods, as reviewed by Roy-Hubara and Sturm [1]. While such methods continue to emerge, we noticed that they consider mainly data-related requirements of the applications (i.e., the data types, their attributes, and the various types of relationships among them), but they usually neglect the functional requirements (i.e., what the users want to do with the data). Moreover, many existing design methods are in their incubation phase, are not widely adopted, and the evaluations of their efficiency and usage are limited.

In this paper, we propose design methods for two types of NoSQL databases: Graph databases and Document databases. These two types have gained remarkable growth in their usage[1] and enable the representation of complex data. Graph databases enable the capabilities of graph theories, allowing more complex queries than the traditional Relational model, mainly since they consider relationships as first-class citizens. Document databases are very popular and widely used (e.g., MongoDB is the most non-Relational database used today [1]).

Nowadays, only a few design methods exist for Graph databases, e.g., [2,3]. In an earlier study, we presented a design method for Graph databases called GDBS [4]. However, the method is solely based on the data-related requirements of the application, which may result in a sub-optimal Graph database schema from the performance point of view. As Graph databases utilize the powers of graph theory in structure and querying, the consideration of query-related requirements when designing such databases may affect both the performance (i.e., execution time) of the sought Graph databases and the complexity of queries posed to the database.

Methods for the design of Document databases also exist, e.g., [5,6]. While they provide important information and insights on the design process, they are not comprehensive and extensive. To create a more comprehensive design method, some of these

* Corresponding author.
E-mail addresses: nro@post.bgu.ac.il (N. Roy-Hubara), sturm@bgu.ac.il (A. Sturm), shoval@bgu.ac.il (P. Shoval).

[1] https://db-engines.com/en/ranking.

https://doi.org/10.1016/j.datak.2023.102149
Received 29 March 2022; Received in revised form 27 January 2023; Accepted 9 February 2023
Available online 16 February 2023

insights should be utilized and extended with all the possible data requirements and functional requirements as well, as the latter may affect both the performance and complexity.

As mentioned, most of the limitations of existing NoSQL database design methods lie in the fact that they mainly refer to data-related requirements, which we claim are insufficient for designing such databases. For example, Huang & Dong [7] conducted several experiments and comparisons with the Graph database system Neo4j. They concluded that it takes more time to execute a more complex query when the number of nodes is large. Thus, it can be deduced that queries and the way they are expressed indeed matter.

In this work, we place the users' requirements of the sought system at the center of the database design process by considering ***not only the data-related requirements but also the functional-related requirements***, which are rarely used in design methods nowadays. In the proposed design methods, the data-related requirements are specified via a class diagram, and the functional-related requirements are specified via queries. With these considerations in mind, we propose two sets of rules for transforming the users' requirements into Graph and Document data logical models. For Graph data models, the rules extend the GDBS method [4]; for Document databases, the method creates a logical data model that considers the different possible ways to implement relationships. We demonstrate how the specification of the requirements can be used by the two methods, and thus, these can be generalized for addressing the design of other database models.

The contribution of this work is as follows:

- We set a common ground for requirements-based database design, regardless of the database type, for future methods. In addition to data-related requirements, we also refer to functional requirements.
- We formulate the design method of a Graph database as a set of rules.
- We formulate the design consideration of a Document database as a set of rules.
- We empirically evaluate the benefits of applying the sets of rules to the performance of the two databases.

The rest of this paper is structured as follows: Section 2 reviews existing methods for designing the two NoSQL databases and analyzes their capabilities. Section 3 presents an example of a certain application that will be used throughout the rest of the paper. Sections 4 and 5 introduce and evaluate the design methods for Graph databases and Document databases, respectively. Section 6 discusses threats to validity of the evaluations. Finally, Section 7 concludes and sets plans for future research.

2. NoSQL database design – state-of-the-art

As the domain of designing NoSQL databases is in its incubation phase, the methods of designing NoSQL databases still rely mainly on best practices and practitioners' experience. However, some studies propose design methods for specific NoSQL databases, and a few propose inclusive design methods for several sub-types of NoSQL databases. Such inclusive methods include the work of Abdelhed et al. [2]. Their work is based on Model Driven Architecture (MDA) which transforms a conceptual model in the form of a UML class diagram into a NoSQL physical model, which is either Column, Document, or Graph. Their method requires the user to decide between several transformation options throughout the process but does not state how to choose or which option is best under what circumstances. They automated their approach for three databases: MongoDB, Cassandra, and Neo4J. Any other databases will require changes to the automation parameters. The method does not address the functional requirements.

In this work, we are interested in the considerations of specific design rather than general transformation; thus in the following sections, we elaborate on graph and document database design methods, respectively.

2.1. Design methods of graph databases

In this section, we review existing design methods for Graph databases. We analyze their capabilities, considering (when applicable) the following criteria: the purpose of the method, its input (i.e., the data and functional requirements), and the evaluation of the method.

The earliest method for designing Graph databases was proposed by Bordoloi and Kalita [8]. It focuses on transforming a Relational database into a Graph database. The method uses a mathematical model to create a Domain-Relationship Diagram (a form of ERD) and creates a graph (but not an explicit model) consisting of the instances from the source Relational database. Thus, no schema or constraints are generated or enforced.

De Virgilio et al. [3] proposed a modeling method based on ERD. In their method, an ERD is transformed into an Oriented ER (O-ER) diagram, which is a directed, labeled, and weighted graph. On top of the O-ER diagram, partitioning is performed based on a set of rules, which split the O-ER into several parts. Finally, based on the partitioned O-ER diagram, a template for a graph database is generated, in which each partition is mapped into a node. The template is kind of a schema for a Graph database. The method was evaluated based on a comparison to the *Sparse* strategy, in which each property value of an object in the ERD is mapped to a node, which is said to be commonly used by Graph database users and generates a significantly large number of nodes. The results of the evaluation indicate that it takes less time to execute queries when applying the proposed design method compared to the *Sparse* strategy. In the latter, there are many more nodes than in the proposed method. Therefore, it seems evident that response time will be faster because of having to traverse significantly fewer nodes. Note that the method does not consider functional requirements.

Daniel et al. [9] proposed a framework called UMLtoGraphDB that translates conceptual schemas expressed in UML into a graph representation. The framework includes rules to transform a class diagram into a graph model, whereas OCL constraints in the class

Table 1
Comparing graph database design methods.

Method	Main concepts	Formalism	Approach (Conceptual, Logical, physical)	Design tool	Automatic transformation	Consideration	Evaluation
Bordoloi and Kalita [8]	Relational model	ERD	Logical	No	No	Data	None
De Virgilio et al. [3]	Relational model	ERD	Logical	Yes	Yes	Data	Comparison to another strategy
Daniel et al. [9]	Classes, relationship & Constraints	UML & OCL	Logical	Yes	Yes	Data	None
Pokorný [10]	Relational model	ERD	Logical	No	No	Data	None
Akoka et al. [11]	Relational model	ERD & 4Vs	Physical	No	Yes	Data	None
De Sousa and Cura [12]	Relational model	ERD	Logical	No	Yes	Data	None
Ghrab et al. [13]	Graph model	GRAD	Logical	Yes	Yes	Data	None
GDBS [4]	Relational model	ERD	Logical	No	Yes	Data	None

diagram are transformed into queries expressed in Gremlin, a graph query language adopted by several Graph database providers. The method is supported by a software tool. This method does not consider functional requirements.

Pokorný [10] discussed general aspects of Graph database modeling. The paper suggests a simple graph database schema based on entities and relationships (a binary ER model) and integrity constraints for the said schema. As in previous works, functional requirements are not addressed.

Akoka et al. [11] proposed a method that considers the four Vs of Big data (variety, velocity, volume, and veracity) for devising a Graph database. The method uses an ERD as a conceptual model, and each component in the model receives information regarding the four V's. The method further applies two sets of rules: the first set enables the translation of the ERD into a logical property graph model, and the second set of rules enables the transformation of the logical property model into a script of Cypher statements. The resulting graph devised from the script contains fictitious data of realistic size according to the volume property of each component in the conceptual model. While 'volume' is the major property considered in this method, the other Vs are less discussed, and it seems that they do not affect the structure of the graph. Here again, there is no consideration of functional requirements.

De Sousa and Cura [12] proposed conceptual and logical models for Graph databases. The conceptual model is in the form of the extended binary entity-relationship model (EB-ER), and a labeled-directed-property based graph is used as the logical model. The method includes an algorithm for transforming the EB-ER to the graph logical model.

Ghrab et al. [13] present a logical model for graph databases called GRAD. GRAD captures traditional modeling concepts, projects them on graphs, and provides the definitions of integrity constraints and graph algebra. The authors implemented a central library implementing their work in a domain and storage-independent manner. While they suggest applications that might benefit from the GRAD model, they describe them briefly.

In previous work, we devised a modeling method for Graph databases called GDBS (Graph database schema) [4]. The method transforms a conceptual schema in the form of an ERD into a Graph database schema in a two-step process (as will be discussed later in Section 3). However, this method did not consider functional requirements, as well.

Table 1 summarizes the analysis we performed. It indicates that all the above methods include some form of modeling where ER is the most prominent. Most of them result in a logical graph model. All methods consider the data-related requirements but neglect the functional requirements. Furthermore, the evaluation is performed to a limited extend. In this work, we propose considering the functional requirements, evaluating the proposed method, and demonstrating the benefit of utilizing it.

2.2. Design methods of document databases

In this section we review existing methods for designing Document databases and analyze their strengths and weaknesses.

Varga et al. [14] propose a method that uses Formal Concept Analysis (FCA), or rather a Relational Concept Analysis (RCA), which extends FCA, as a data model. According to the authors, "the objective of RCA is to build a set of lattices whose concepts are related by relational attributes, similar to UML associations". The method uses ER as a conceptual model that is transformed into a Relational Context Family, which is the structure of data within the RCA. The method refers to the two different relationship types in Document databases, which is very important in document modeling. Furthermore, it considers the cardinality between entities in order to choose the right type of relationship. When the two relationship types are possible, they do not suggest other means of distinction. Furthermore, the method does not consider functional requirements.

In another paper, Varga et al. [15] suggest modeling MongoDB data structure as conceptual graphs (CG). In their work they demonstrate a conceptual graph structure for the database and its queries. Their reference to queries is only to their representation with CG, and not to their effect on the design itself.

De Lima and Dos Santos Mello [16] suggest a method based on EER model and queries workloads. The workload information is estimated over the conceptual schema, i.e., the size of each construct in the EER model. This is used to estimate the number of accesses for each construct based on access patterns, which in turn is applied in choosing between different relationship types. The method introduces essential concepts such as the use of workloads. The transformation rules and functions are based on the cardinality of the constructs in the relationships, but this concept is not explained. The method is evaluated based on a case study in which the authors compare their method with and without the workload information.

Table 2
Comparing document database design methods.

Method	Main concepts	Formalism	Approach (Conceptual, Logical, physical)	Design tool	Automatic transformation	Consideration	Evaluation
Varga et al. [14]	Relational model	RCA	Logical	No	Yes	Data	Performance
Varga et al. [15]	Graph model	Conceptual graph	Physical	No	No	Data and query representation	None
De Lima and Dos Santos Mello [16]	Relational model	ERD	Logical	No	Yes	Data, volume, and workload	Case study
Imam et al. [5]	Relationship classification		Logical	No	Yes	Data	Experiments
Herrero et al. [17]		Graph	All	No	Yes	Data and workload	None
Shin et al. [6]	Class Model	UML	Logical	No	Yes	Data	Example

Imam et al. [5] further refine the notion of cardinality and suggest a way to handle them. The concept of "many" is broken down into several levels: few, many, and squillions. While the paper mainly suggests these new cardinalities for the conceptual model, it also demonstrates important experiments that assist in choosing the right kind of relationship in the database. The experiments lead them to recommend the types of relationships. Nevertheless, they did not consider functional requirements.

Herrero et al. [17] present a method for designing NoSQL databases based on traditional design approaches. The method consists of seven steps spanning the three phases of conventional design (i.e., conceptual, logical, and physical). The fourth step of their method addresses the issue of merging nodes to improve performance. As it is very similar to the issue of embedding vs. referencing in Document databases, it sheds light on the essential matters in Document database design, that is how to deal with different relationship types and their cardinalities.

Shin et al. [6] demonstrate the design of a Document database based on a UML class diagram. Their transformation is relatively simple: a class is transformed into a document collection, an attribute is transformed into a property of a document, and an association is transformed into a relationship between documents. The method is simple but does not consider more complex relationships in a UML class diagram, such as hierarchies and aggregations. Moreover, it does not prescribe how to choose the type of relationship between documents, which is a primary issue in Document database design.

Table 2 summarizes the analysis we performed. In the case of document databases, it seems that each method refers to a specific view neglecting the need for a comprehensive view or a design process. Most methods refer to the selection of relationship types and usually aim at a logical document database model. Although guide- lines and rules exist, design tools hardly exist. The most prominent gap is the lack of reference to functional requirements.

3. Example of an application

Before presenting the design methods in Sections 4 and 5, in this section, we provide an example of an application that will be used later to explain and demonstrate the methods. We use an IMDb-like application, which manages data about watch items: a general term for movies, series, and episodes. For each watch item, the application manages data about its directors, producers, and actors and their roles in the watch item. The application also stores information for each watch item, such as goofs (mistakes), trivia information, and quotes. A user of this application may log in to rate a watch item. Fig. 1 presents the conceptual model – a UML class diagram – of this application, which represents the users' data-related requirements. We choose UML for that purpose for the following reasons: (1) It is the standard modeling language; (2) The class diagram of UML is well-known and easy to use; (3) It is commonly used by developers; (4) As the database is part of the development process, we aim at using the same language for all artifacts; and (5) We use only a subset of the UML class diagram notation, which facilitates the definition of struct types, relationships, and properties.

In addition to the conceptual model of the application, we present the users' functional requirements as a list of queries — see Listing 1. The list consists of ten queries, all of which are of "select" type, though queries of "update" type may also be considered (yet, usually, these are simpler).

The queries are written in an SQL-like language adopted from Mior et al. [18].

- **Query 1**. RETURN password FROM User WHERE username = ?
- **Query 2**. RETURN ALL FROM WatchItem WHERE title = ?
- **Query 3.** RETURN ALL FROM WatchItem, Person WHERE rel.role like ?
- **Query 4**. RETURN rec.title FROM WatchItem as rec, WatchItem as org WHERE org.Genre = rec.Genre and type(org)=type(rec) and rec.releasedate BETWEEN org.releasedate += 10
- **Query 5.** RETURN Person.All FROM Watch Item, Person WHERE title = ? and rel = Actor
- **Query 6.** RETURN Person.All FROM Person WHERE gender = ?
- **Query 7.** RETURN e1.All FROM Episode as e1, Episode as e2 WHERE e2.season=? and e2.number=?. and e1.season= e2.season and e1.number = e2.number+1
- **Query 8.** RETURN Genre.All FROM User, Rate-Person-Movie,Rate, WatchItem, Genre WHERE username = ? and numericrating = 5
- **Query 9.** RETURN p1.ALL p.2 ALL FROM Person as p1, WatchItem, Person as p2 WHERE Person.name = ?
- **Query 10.** RETURN WatchItem.All FROM WatchItem, Person WHERE Person.name = ?

Listing 1. A List of queries for the IMDb application

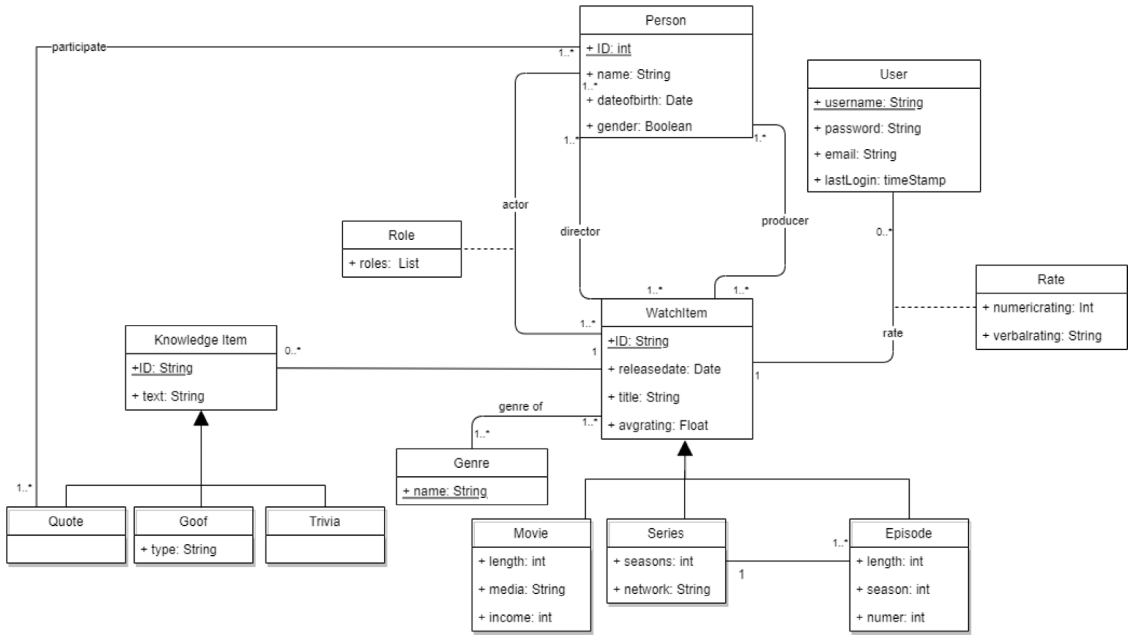


Fig. 1. The conceptual model of the IMDb application.

In the two next sections, we present the design methods for Graph and Document databases. The two methods use the same input: a conceptual model in the form of a UML class diagram, which represents the data-related requirements, and a list of queries in an SQL-like syntax, which represents the ***functional requirements***, as described above. The list may change throughout the lifecycle of the application (and thus, the design might change as well). In that case, the impact of schema evolution (e.g., [19]) should be considered as well. Yet, this is out of the scope of this paper.

For each database type, we define a specific set of design rules for transforming these inputs to the respective NoSQL database logical schema (i.e., design).

4. Design method for graph database and its evaluation

In the following section, we introduce the method for graph database model design and its evaluation. The method is demonstrated with the example introduced in Section 3.

4.1. Fundamentals of graph databases

A Graph database logical model consists of the following components:

- ***Node*:** A node represents an entity in the real world; it has a label (a name) and properties, including a key property that enables its unique identification.
- ***Edge*:** An edge represents a binary relationship between two nodes. As defined in most graph databases [20], an edge has a direction, meaning it has a start node and an end node. An edge may also have properties and a label.
- ***Property*:** As said, both nodes and edges may have properties. Properties may have constraints, such as (a) Key: each node has a key, which may be one property or a combination of several properties; (b) Not Null: the property must have a value for all instances of the node or edge; (c) Set: the property may have many values.
- ***Cardinality constraints*:** Cardinality constraints restrict the number of nodes that may participate in a specific edge type, ranging from at least 0 or 1, to at most 1 or many (not restricted).

In the design process, we will use the following notations:

- Cl = set of all classes in the conceptual model.
- R = set of all relationships in the conceptual model.
- A = set of all attributes in the conceptual model: A(Class) = set of all attributes of a class, V_a = values of an attribute.
- Q = set of all queries that represent the functional requirements.
- F = set of all filtering objects (aka where clauses) in Q; Co = conditions in a filter.
- $\prec$ = a hierarchy of classes; type ($\prec_i$) = type of the hierarchy (cartesian product of complete and disjoint hierarchies).
- # = the number or size of a thing (i.e., #Q = number of queries)

4.2. Transformation rules that consider the data requirements

In an earlier study [4], we introduced a method to design a Graph database schema — GDBS. The method used ERD as a conceptual model, which represents the data-related requirements. In this study, we use UML class diagram instead of ERD for the reasons we specified before.

The transformation of the class diagram to a Graph database schema is a two-phase process. First, we transform some special constructs in the class diagram into binary relationships. Second, we transform the adjusted class diagram into a Graph database schema. Note that at this stage, the functional requirements are neglected.

4.2.1. Adjusting the original class diagram

In this phase, we transform all types of hierarchy relationships, aggregation, composition, and association classes. This step is derived from our previous work [4]; thus, we describe the rules briefly.

Hierarchy (inheritance) Relationships

For a hierarchy (aka is-a) relationship, we distinguish between two cases:

- Removing the sub-classes of the hierarchy and moving their attributes and relationships to the super-class, adding to it a new property named 'type' to enable distinguishing between the different sub-types. This mapping rule is applied in cases where the inheritance relationship has **neither** a complete (total-cover) constraint, **nor** the disjoint constraint, meaning that there may be an object of a super-class that is not of one of the sub-classes, or that an object may belong to many sub-classes.
- Removing the super-class and moving all its properties and relationships to each of its sub-classes. This mapping rule is applied in cases where there are both complete **and** disjoint constraints between the sub-classes, meaning that all objects of the super-class belong to one sub-class only. Therefore, there is no need to maintain the super-class.

Formal definition of the rule: $\exists \left(c_1, \ldots, c_n\right) \prec_i c_{super}$

1. If type($\prec_i$) = () or (disjoint) or (complete) $\rightarrow Cl = Cl - \left(c_1, \ldots, c_n\right)$ *and* $A\left(c_{super}\right) = A\left(c_{super}\right) \cup \forall i \quad A\left(c_i\right)$ *and* $R = R - \forall i \quad R\left(c_i, C_j\right) + R\left(c_{super}, C_j\right), C_j = group\ of\ classes\ which\ are\ related\ to\ \left(c_1, \ldots, c_n\right)$
2. If type($\prec_i$) = (disjoint, complete) $\rightarrow Cl = Cl - \left(c_{super}\right)$ *and* $A\left(\left(c_1, \ldots, c_n\right)\right) = A\left(c_{super}\right) + A\left(\left(c_1, \ldots, c_n\right)\right)$ *and* $R = R - R\left(c_{super}, C_j\right) + R\left(\left(c_1, \ldots, c_n\right), C_j\right), C_j = group\ of\ classes\ which\ are\ related\ to\ c_{super}$

Aggregation and Composition Relationships

In a UML class diagram, aggregation and composition relationships are binary associations between a 'whole' class and its 'parts' classes. Composition is a stronger type of aggregation, in which an object in the 'part' class may be associated with only one object of the 'whole' class, while in aggregation, an object of a 'parts' class may be associated with several objects of the 'whole' class.

Hence, the rule is to transform both types of relationships based on the above two cases: an aggregation is transformed into many-to-many relationship, whereas composition is transformed into a one-to-many relationship, where the cardinality of the relationship at the whole-class is 0,1 (that is: min. 0 and max. 1).

Association Classes

A UML class diagram may include association classes. An association class belongs to a relationship between two classes. In our example (Fig. 1), there are two association classes: one, named Rate, which belongs to the relationship between classes User and Watch Item, and a second, named Role, which belongs to the relationship between classes Person and Watch Item. Following GDBS in the UML class diagram, an association class will be transformed into regular attributes of the relationship.

4.2.2. Mapping the adjusted class diagram to a graph database schema

In this stage, the components of the adjusted class diagram are mapped to a Graph database schema. Here are the rules:

1. **Mapping classes to nodes**. Each class is mapped to a node, and its properties become the node's properties.
2. **Mapping relationships to edges**. Each relationship between entities is mapped to an edge connecting the respective nodes. The edges are directed, distinguishing between the start and end nodes of each edge.
3. **Mapping cardinality constraints**. The cardinalities of each relationship are mapped to the edges. (It should be noted that such constrains are not part of ordinary Graph databases).

Applying the GDBS method for the IMDb application results in a Graph database diagram that is shown in Fig. 2. Note that the symbols used in this diagram are like class diagram symbols, only that here round rectangles denote nodes, arrows denote edges, and the arrowheads indicate the directionality of the edges. Each round rectangle (node) includes the node's possible set of properties.

To keep the presented schema concise, we have unified some of the edges between nodes: Note the edges involving the nodes *Episode*, *Series*, and *Movie;* these nodes are connected to *Goof*, *Trivia*, and Quote: a total of nine edges should be shown here (i.e., *Movie – Quote, Movie – Goof, Movie – Trivia, Episode – Quote, Episode – Goof, Episode – Trivia, Series – Quote, Series – Goof, Series – Trivia*). We unified the said edges, while only the heads and tails of the edges were split. To avoid similar clutter, some edges were omitted and written in words only. The full schema definition of the Graph database should include **all** the omitted/unified edges.

Note that the *WatchItem* class in the class diagram was removed (i.e., it did not become a node), due to the relevant rule, while all its properties are included within its sub-classes. However, the queries in the following will still use the general term *WatchItem*, which may refer to any of its children (*Movie, Series, or Episode*).

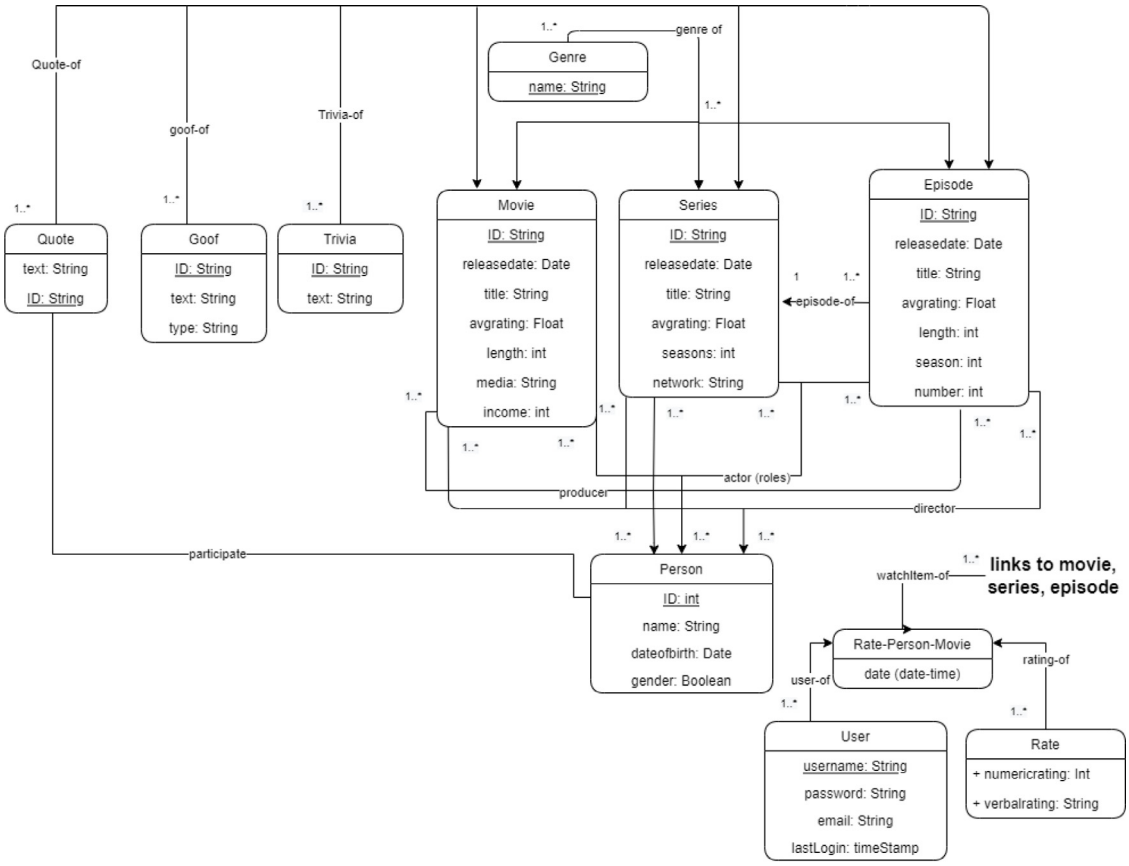


Fig. 2. Graph database diagram for the IMDb application after applying the transformation rules.

4.3. Transformation rules that consider functional requirements

Next, we consider functional requirements that are related to database manipulation, namely, data update and retrieval operations, which are expressed by queries that perform Select, Update, Insert and Delete operations. In the Relational model, considering such requirements may result in adding definitions of secondary indexes for specific attributes, adding assertions, triggers, stored procedures, or even de-normalizing some tables. Considering such requirements in Graph databases may result in splitting or unifying nodes, adding edges, indexes, etc. Considering functional requirements may also result in data duplication (e.g., duplication of a property in different nodes). It is assumed that the consideration of functional requirements would generate a database that provides better response time, as it would fit better the users' needs.

In the following, we introduce a set of rules, based on the functional requirements (namely the queries) aimed at improving the Graph database schema created solely based on the data-related requirements. We do not claim that the proposed rules are comprehensive; further research may identify more rules. Nevertheless, we will show how rules that are based on functional requirements may change the Graph database schema and result in better performance.

In devising these rules, we rely on knowledge and experience gained by professionals, designers, developers, and Graph database providers (e.g., Neo4J[2]). While most of that knowledge is adopted from Robinson et al. [20], we found further examples and extensions in forums, question sites (such as Quora[3] and Stack Overflow[4]), and tutorials.

In the following, we describe the proposed set of rules, along with examples.

Adding Generic Edges

In the class diagram of the IMDb application (Fig. 1), there are three relationships between class *Person* and each of the *WatchItem* sub-classes: *actor*, *producer,* and *director*. According to a previous rule, these relationships were transformed into three edges between each of them and *Person*. Nevertheless, users may be interested in checking the existence of a relationship between *Person* and any of the sub-classes of *WatchItem*. Such a case can be revealed by examining the functional requirements. See, for example, Query 10

[2] https://neo4j.com/.
[3] https://www.quora.com/.
[4] https://stackoverflow.com/.

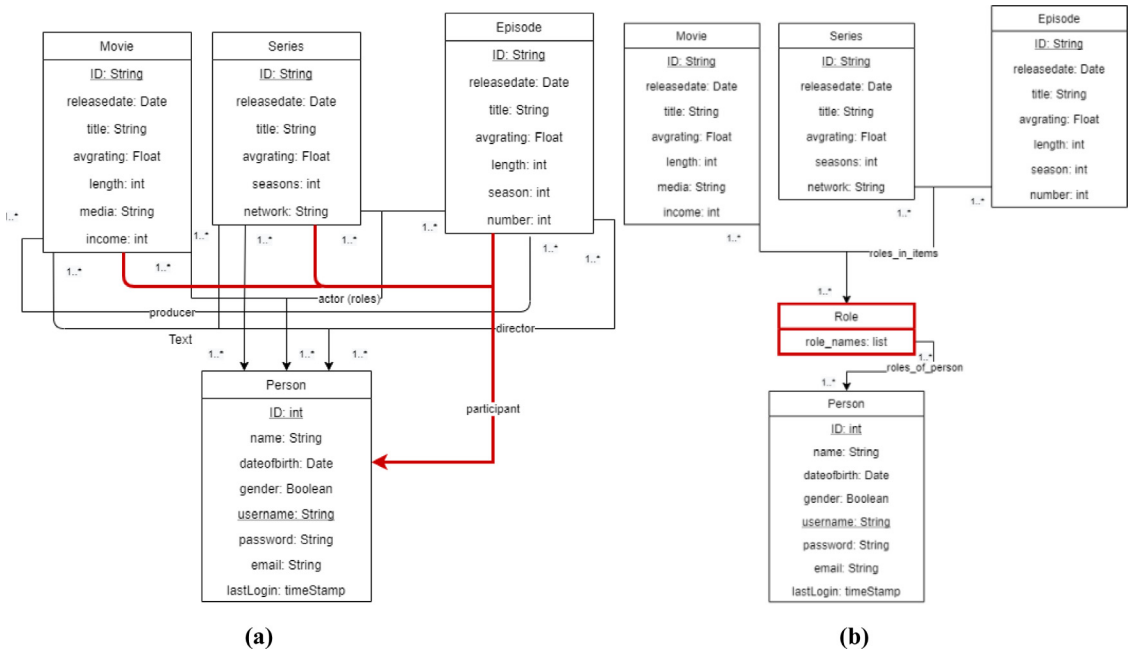


Fig. 3. The Graph schema: (a) after applying Rule G1, (b) after applying Rule G2.

in Listing 1: "Return all projects in which a person was involved in any capacity". In such cases, adding a generic edge between the nodes *Person* and the different *WatchItems* (that can be labeled "participant") would make this query more efficient.

Rule G1: In cases where queries involve any type of relationship between two classes (e.g., a query contains "or" operators between relationship types), a generic edge should be added to the Graph database schema (in addition to the existing edges).

Formal definition:

$$c_1, c_2 \in q_i \; and \; \#R(c_1, c_2) > 1 \; and \; \#(R(c_1, c_2) \in q_i) > 1 \rightarrow R\left(c_1, c_2\right) = r_{new}\left(c_1, c_2\right) \cup R(c_1, c_2)$$

Following this rule, we add the generic relationship "participant" from class Person to its sub-classes — see Fig. 3a.

Adding Intermediate Nodes

In some cases, intermediate nodes may be added instead of edges between nodes. For example, in the IMDb application, the relationship *actor* between *Person* and *WatchItem* has an attribute in the form of a list (i.e., roles). It may be difficult to pose queries involving this attribute, either because the query may be too complex, or because the processing time of the query may be too long (since finding a specific role requires filtering on the *Person*, finding the specific relationship to the watch item and then searching the list of roles. For example, the query "Find all roles of *Tatiana Maslany* in *Orphan Black*" requires the system to find the Actor node of *Tatiana Maslany*, then the edge of the said node to the node of *Orphan Black*, and finally return all the roles on the said edge). A similar example is when a filter is applied to an attribute that has a limited number of values, such as *Gender*. Based on best practices, if a query includes such a filter, it may be more efficient to define this attribute as a node rather than to query and filter on the attributes' value.

Another example for adding an intermediate node is modeling facts as nodes, as Robinson et al. [20] stated: "When two or more domain entities interact for a period of time, a fact emerges". The outcome of such interaction should be modeled as an intermediate node. A common example is Email: while Email can be modeled as an edge between two person nodes (Person A sent an Email to Person B), Email is referred to as a fact since it is an interaction between two nodes and contains important information such as time and type of addressee (CC, BCC), and the content of the message. Thus, we define the following two rules:

Rule G2: If there are queries that require filtering on a relationship attribute, the relationship is probably a fact and thus should be transformed into a node.

Formal definition:

$$a_i \in r_i \; and \; F\left(a_i\right) \in q_i \rightarrow Cl = c_{new}(a) \cup Cl, R = R \cup (R\left(c_{start\ of\ r_i}, c_{new}\right) + R(c_{new}, c_{end\ of\ r_i})) - r_i$$

Rule G3: If there are queries that require filtering on a limited set of attribute values, this attribute should be transformed into a node.

Formal definition:

$$V_{a_i} = \left(v_1, v_2, \ldots, v_n\right) where\ n < x, \; and \; F\left(a_i\right) \in q_i \rightarrow Cl = c_{new}(a) \cup Cl, R = R \cup R\left(c_{new}, c_{a_i}\right).$$

x *is determined by the user.*

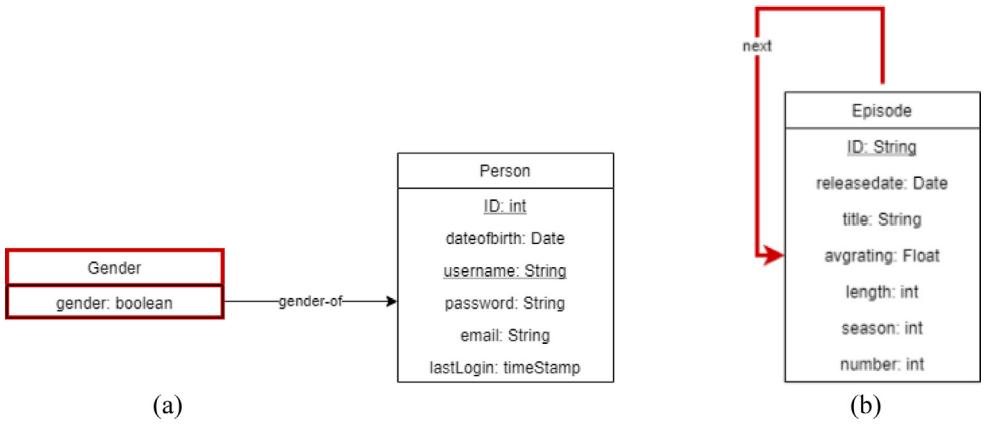


Fig. 4. The Graph schema: (a) after applying Rule G3, (b) after applying Rule G4.

For both rules, the label of the added node will be the attribute's name, while its instances will be its possible values. For example, following Query 3: "*Return All From WatchItem, Person Where rel.role like ?*", in Fig. 3b the new node is labeled role. Following Query 6: "*Return Person, All From Person Where gender = ?*", results in a new node labeled Gender — see Fig. 4a.

Adding a Relationship for Sequences

In the process of modeling, we many times tend to ignore sequences. For example, the IMDb application contains information about episodes of a series: Episodes of a series are a sequence, a fact that is not addressed in the model. There may be queries that refer to such a sequence. For example: "return the next episode". For such a case, we create a unary edge in node *Episode* named "next".

Rule G4: Instances of a node that are considered as a sequence should be linked by an edge that will make them a linked list. The direction of the link will be based on the queries (forward, backward, or bi-directional). If no such query exists, then by default, only a forward edge will be added to the schema.

Formal definition:

$$c_1, c_2 \in q_i \; and \; type\ (c_1) = type\,(c_2)\ \; and \; c_1, c_2 \in co_i \; and \; (+,-) \in co_i \rightarrow R\,(c_1, c_2) = r_{next}\,(c_1, c_2) \cup R(c_1, c_2)$$

For example, *Return e1.All From Episode as e1, Episode as e2 Where e2.season=? and e2.number=? and e1.season= e2.season and e1.number = e2.number + 1*. This query will return the next episode. As shown in Fig. 4b, the relationship *next* was added to the schema to support the sequence of episodes.

Adding Implicit Relationships

Sometimes queries may discover implicit relationships between classes, which are not present in the class diagram. For example, the IMDb application may be used for recommending watch items by users based on genres, actors, etc. For example, the application may recommend a new watch item from a frequently watch genre:

RETURN rec.title FROM WatchItem as rec, WatchItem as org, Genre, User Where User.username = ? and count(Genre.name) > 10

This knowledge is not explicated in the conceptual model but is implicit. In such a case, an explicit edge should be defined between the relevant nodes.

Rule G5: Paths in the functional requirements with implicit meaning should be changed to explicit edges.

Formal definition:

$$Let\ Path = p\,(c_1, \ldots, c_n)\,, \forall_{i=2..n} \exists r_i\,(c_{i-1}, c_i)$$

$$\exists p, \forall c_i \in p \; and \; c_i \in q_i \rightarrow R\,(c_1, c_n) = r_{new}\,(c_1, c_n) \cup R(c_1, c_n)$$

For example: in Query 8, which represents a recommendation based on genres: "*Return Genre.All From User, Rate-Person-Movie, Rate, WatchItem, Genre Where username = ? and numeric rating = 5*", a new edge is created between Genre and User — see Fig. 5.

4.4. Evaluation of the graph database designed with considering functional requirements

We performed two experiments with two different applications to evaluate the performance of a Graph database designed considering the functional requirements compared to a Graph database designed without considering these requirements. All relevant materials are available in an open Dropbox folder.[5]

[5] https://wwwwww.dropbox.com/sh/uzu1dnnsj3u6m8d/AABMfxTpBfEaQb6yk5gnHW6ca?dl=0.

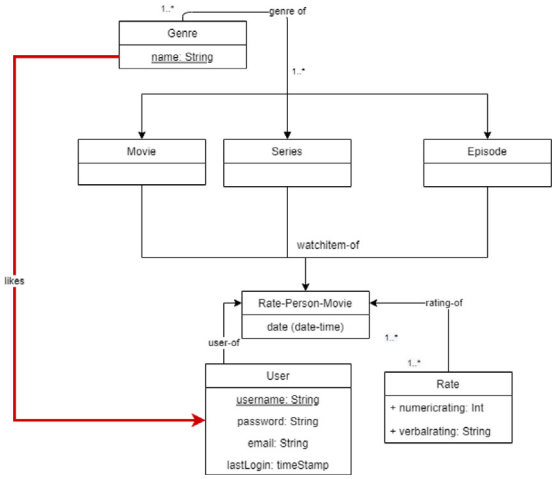


Fig. 5. The Graph schema after applying Rule G5.

Table 3
Number of labels in the original IMDb graph database.

Label	Count	Label	Count
Person	1907	Episode	921
Genre	21	Trivia	18 768
Movie	100	Quote	11 913
Series	9	Goof	7679

4.4.1. Experiment hypothesis, variables, and technical set-up

We conjecture that there is a difference in the performance of a Graph database designed with rules that consider the functional requirements compared to a database designed without considering them. Therefore, we set the following statistical hypothesis:

$$H_0 : Performance_{BL}^{Factor} = Performance_{RULE}^{Factor}$$

BL stands for baseline system, i.e., without considering the functional requirements rules, and RULE stands for the systems that consider them (1–5, or all of them).

We experimented with a Neo4J Graph database (Neo4J 4.0.4 on Neo4j desktop), as it is the leading graph database provider nowadays, installed over a virtual machine with 8 GB RAM, 250 GB disk size, and Windows-OS. The factors that we measured in the experiment were **query execution time** and **DB hits** (an abstract unit that counts the number of requests sent to the storage engine). Several actions trigger DB hits, such as getting a node by ID or a property of a node or a relationship. The full list of actions can be found in the Neo4j documentation.[6]

Note that when applying the rules to each application, the database gets larger with more nodes, labels, relationships, and relationship types. It makes sense to assume that the execution time of queries may become slower as the data volume increases. The evaluation we performed checks whether the increased volume of data resulting from applying the rules negatively affects the query performance or improves it.

4.4.2. Evaluation of the IMDb application

For the IMDb application, we performed the following experiment: we created six databases: the original database (created from the original GDBS), four databases, where each is built by applying one of the functional-related rules G1 to G4 separately (rule G5 is not applicable in the example), and a database that applies all four rules.

The data for the databases were loaded from IMDb application via Python code with IMDbPY,[7] a package in Python which accesses IMDb web server and retrieves data. Since the data in IMDb is limited to publicly available information, including Persons, Movies, Series, and Episodes, we created a sample database without Users and Ratings.

We populated the databases as follows: First, with IMDbPY we created the original database, a snippet of IMDb, which contained 41,318 nodes and 55,147 relationships. The number of entities is shown in Table 3. This database was cloned for each rule. In each of the four clones, we applied one rule, while in the last clone, we applied all four rules and adjusted the data to fit the new schema. Table 4 presents the descriptive statistics of the six databases.

[6] https://neo4j.com/docs/cypher-manual/current/execution-plans/#:~:text=unique%20end%20nodes.-,3.,of%20this%20storage%20engine%20work.
[7] https://github.com/alberanid/imdbpy.

Table 4
Summary of the IMDb graph databases.

Database	Number of nodes	Number of labels	Number of relationships	Number of relationship types
IMDb-Like Baseline	41 318	8	55 147	8
Applying Rule G1	41 318	8	68 473	9
Applying Rule G2	42 837	9	66 989	10
Applying Rule G3	41 320	9	57 053	9
Applying Rule G4	41 318	8	56 008	9
Applying All 4 Rules	42 839	10	83 082	13

To test the hypothesis, we run the queries in Listing 1. Only eight out of the ten queries are relevant (Queries 1 and 8 were omitted since they deal with the User class, which is not included in the database.). The queries were expressed in Cypher, the Neo4j query language. Before running the queries, they were examined for correctness (i.e., that they returned the expected results) in all six database versions. Lastly, all queries were run after arriving at a stable cache.[8]

Results and Analysis

We tested the time of execution of the queries and the number of DB hits. Four of the eight queries were addressed by the four rules, i.e., the rules were applied based on these queries. The other four queries should not be affected by applying the rules. While after applying the rules, the databases become larger than the original, the rules, and thus the design, address the application's functional requirements better. Consequently, we expect to see better results in these measures.

Table 5 presents the results. The gray-colored cells indicate the relevant query for the rule of the database, i.e., the rule that was applied due to the said query and therefore had to be adjusted (i.e., to be written differently in Cypher) based on the new database design. For example, Rule G1 is applied due to Q10, which searches for relationships between the classes *Person* and *WatchItem*. Therefore, the cells that cross both Rule G1 and Q10 are gray-colored.

The results present major improvements when moving from the IMDb original database (the "baseline") to the databases that consider functional requirements, i.e., that apply the above rules. Applying the rules reduces the number of DB hits by 17% and the execution time by 53%. In queries affected by the rules (Q3, Q6, Q7, and Q10), the total number of DB hits decreased, while the number of DB hits increased in the rest of the queries. We conjecture that the reason for this increment is because of the increase in the size of the graph. The total number of DB hits and execution time improved when applying the rules, as can be seen on the right-hand side of the table.

To examine the significance of these improvements, we tested the effect of the application of each rule individually and compared it to the baseline database. We performed paired t-tests for both DB hits and execution time. We found out that for three out of the four rules, the application of a single rule did not improve these measures significantly (p-value > 0.05), while applying rule G4 improved the execution time significantly (p-value = 0.042). This means that the application of each rule alone, other than rule G4, did not improve the measures significantly.

We then compared the baseline databases to the final database in which all four rules were applied. In this case, while the improvement in the DB hits was not statistically significant, the improvement of execution time was statistically significant (p-value= 0.045). To assess the significance of the improvements, we also calculated Hedges's g[9] for a small number of samples (<20). We found out that the rules caused a medium effect (0.682, using average variance[10]), i.e., the change in execution time is indeed significant and matters.

We also examined whether the application of the rules resulted in performance reduction due to non-relevant queries. For both measurements of time and DB hits, the difference between the paired groups was not statistically significant, i.e., the newly designed databases did not cause deterioration in the database performance.

4.4.3. Evaluation of the northwind database

Northwind[11] is a sample database used by Microsoft to show and learn its programs (Microsoft access, management studio) capabilities. Since this is a well-known database, we decided to use the set of provided queries[12,13] for our purpose with some minor changes. In Fig. 6, the said class diagram is presented, while the relevant queries appear in Listing 2.

In the Northwind evaluation, we created two databases: the first is a baseline database based on the schema and the transformation rules based on data-related requirements only; the second is after utilizing the functional-related transformations rules. Based on the queries in Listing 2, we applied two rules: Rule G3 was applied on the attribute "discontinued" in the product nodes, and rule G5 was applied twice: new relationships between *Category* and *Order*, and *Category* and *OrderDetails*, were created. The size of the baseline database is 3.28 MiB, while the size of the transformed database is 4.36 MiB

[8] https://community.neo4j.com/t/query-execute-time-varies/24556.

[9] https://www.statisticshowto.com/hedges-g/.

[10] https://www.real-statistics.com/students-t-distribution/paired-sample-t-test/cohens-d-paired-samples/.

[11] https://www.outsystems.com/forge/component-overview/7058/northwind-db#:~:text=The%20Northwind%20database%20is%20a,speciality%20foods%20export%2Fimport%20company.&text=Very%20good%20example%20to%20test%20and%20practice%20data%20queries.

[12] https://www.geeksengine.com/database/problem-solving/northwind-queries-part-1.php.

[13] http://www.geeksengine.com/database/problem-solving/northwind-queries-part-2.php.

Table 5
The IMDb application: DB hits and execution time (in ms) for different queries and Graph databases.

Data-base	Perfor-mance	Q2	Q3	Q4	Q5	Q6	Q7	Q9	Q10	Sum
Base-line	**DB Hits**	135665	36463	16990	4686	6886	25357	9471	8001	235518
	Time	44	39	55	1	7	11	10	26	193
Rule G1	**DB Hits**	135665	39642	17012	4699	6886	14889	11417	**8002**	238212
	Time	22	28	21	5	4	16	7	**11**	114
Rule G2	**DB Hits**	140222	**4562**	17009	4696	6886	14685	9523	8003	205586
	Time	20	**19**	18	20	3	11	6	15	112
Rule G3	**DB Hits**	134345	38215	16990	4686	**2767**	18321	9203	6677	231204
	Time	17	35	22	3	**4**	11	11	6	109
Rule G4	**DB Hits**	136586	36463	18332	5609	6886	**10108**	9471	8001	231456
	Time	36	35	34	5	5	**7**	5	9	136
All 4 rules	**DB Hits**	139823	**4562**	18364	5632	**2767**	**6125**	11243	**6677**	195193
	Time	32	**5**	27	<<1	**3**	**7**	17	**<<1**	91

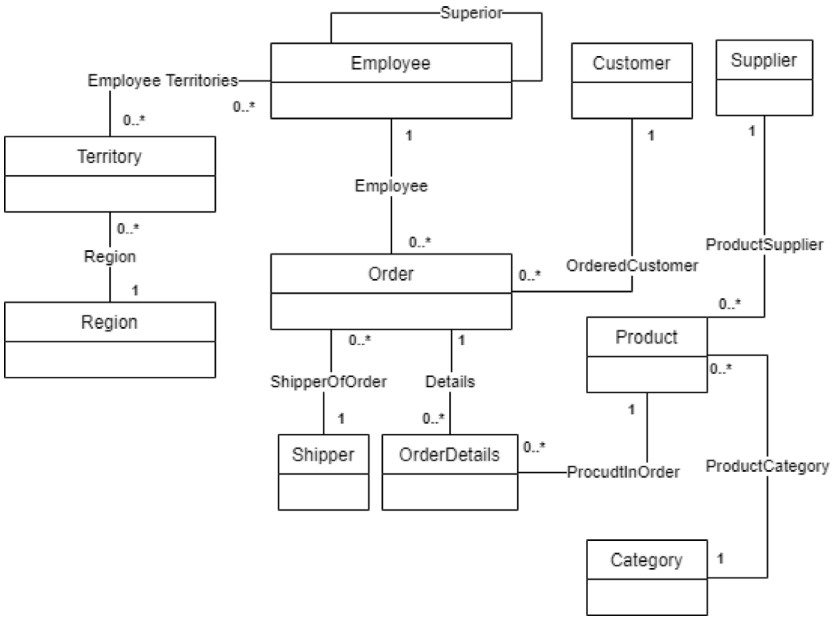


Fig. 6. The class diagram of the Northwind database.

As in the previous application, we examined the queries in the listing. Four of the ten queries were affected by the rules. As in the first evaluation, the queries were translated to Cypher, examined for correctness, and run after arriving at a stable cache.

- **Query 1**. Calculate order cost. (RETURN OrderID, MATH(Quantity, Discount, UnitPrice) FROM Order, OrderDetails)
- **Query 2**. Categories per order. (RETURN OrderID, COUNT(Category)) FROM Order, OrderDetails, Product, Category)
- **Query 3.** Sales per employee. (RETURN EmployeeID, MATH(Quantity, Discount, UnitPrice) FROM Employee, Order, OrderDetails)
- **Query 4**. List of product details. (RETURN CategoryID, SupplierID, ProductName FROM Product, Category, Supplier)
- **Query 5.** Current product list. (RETURN ALL FROM Product WHERE Discontinued = 0)
- **Query 6.** Sales per category. (RETURN CatgoryID, MATH(Quantity, Discount, UnitPrice) FROM Order, Product, Category)
- **Query 7.** Sales per category and product. (RETURN CatgoryID, ProductName, MATH(Quantity, Discount, UnitPrice) FROM Order, Product, Category)
- **Query 8.** Top ten most expensive products.(RETURN ALL FROM Product ORDER BY UnitPrice)
- **Query 9.** Current products per category. (RETURN CategoryID, ProductName, QuantityPerUnit FROM Category, Product WHERE Discontinued = 0)
- **Query 10.** Suppliers and customers cities. (RETURN city FROM Supplier, Customer)

Listing 2. A List of the Northwind System queries

Results and Analysis

Table 6 presents the results for the Northwind application. The results show minor improvements when moving from the baseline database to the final database. Applying the rules reduces the number of DB hits by 6% only, but the execution time was reduced by

Table 6
The Northwind application: DB hits and execution time (in ms) for the different queries and Graph databases.

Database	Performance	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Sum
Baseline	DB hits	36 941	37 002	28 846	3091	976	24 179	24 378	362	3870	449	160 094
	Time	15	31	15	15	15	15	16	≪1	≪1	31	153
Final	DB hits	38 849	**24772**	30 754	3122	**629**	**23 811**	24 460	316	**3422**	449	150 584
	Time	39	**15**	15	≪1	**≪1**	**16**	16	≪1	**≪1**	16	117

Table 7
Summary of the "big-data" IMDb Graph Databases.

Database	Number of nodes	Number of labels	Number of relationships	Number of relationship types
Large Baseline	19,416,764	6	69,088,838	7
Applying All 4 Rules	85,547,397	8	186,291,356	12

Table 8
The IMDb application: DB hits and execution time (in ms) for different queries and Graph databases.

		Q2	Q3	Q4	Q5	Q6	Q7	Q9	Q10	Sum
Large Baseline	**DBHits**	99347595	14453015	6027527	15212624	9291	19072165	58461867	58455660	271039744
	Time (ms)	13952	8233	6398	15687	11	4266	9664	9268	67479
Applying All 4 Rules	**DBHits**	311464278	**115713**	6027527	15212624	**2055**	**689463**	**53057296**	**53050985**	439619941
	Time (ms)	37035	**1413**	2177	18365	**6**	**162**	**8462**	**7677**	75297

24%. As in the IMDb experiments, we performed paired t-tests for both DB hits and execution time. While none of the differences were statistically significant (p-value > 0.05), there were some improvements, especially in the time of query execution. Note that the Northwind database is relatively small (~10,000 nodes); we assume that a larger database would show much more significant improvement. Since none of the measures were improved significantly, we did not calculate the Hedges's *g*.

4.4.4. Discussion of the results

First, in both evaluations, we worked with a small set of queries. We assume that the more queries affected by the rules, the more significant the impact would be, since more specifically tailored changes will be applied.

Second, in the first experiment, while execution time improved significantly, the number of DB hits was improved only to a limited extent. This is probably due to the way the data is stored in neo4j system. For example, in Neo4j, the properties of a node are stored as a linked list, while a node stores a pointer to its first property.[14] In such cases, to access a node's property, a traversal of the linked list is required. More DB hits will be required if the sought property is at the end of the list. Therefore, new queries that required more traversal on properties did not improve the number of DB Hits. However, the time improved as the required actions for the new design were less time-consuming.

Third, we assume that the reason for the significant improvement of execution time also lies in the way data are stored in neo4j: Neo4j and other Graph databases use an index called index-free-adjacency. As the name implies, each node directly references its adjacent nodes. This makes traversing neighboring nodes quite easy and timesaving. Therefore, changes to the database that require query traversing on neighboring nodes (instead of properties, for example) improved the response time.

4.5. Graph database scalability analysis

In order to test the method in a big-data environment, we used IMDb, public datasets.[15] The files were transformed into CSV files and then loaded to Neo4J with the same settings as in the experiment in Section 4.4. As the files were very large, we used Neo4J capabilities for periodic commits and iterations. Since we used the datasets that are open to the public, some data was not included, such as goofs, quotes, and trivia. We decided to generate data regarding quotes in order to make the data aligned with the requirements.

We created two databases: the original and one after applying all the rules. The characteristics of the two databases appear in Table 7.

On these two databases, we execute the relevant queries (as discussed in Section 4.4.2) and check the results for the significance of the improvements by performing paired t-tests for both DB hits and execution time. Table 8 presents the results in the same way shown in Section 4.4.2, in which gray-colored cells indicate queries that were changed due to one of the rules. In this case, after further analysis, we noticed that another query would benefit from rule application — query 9, in which we used the generic relationship.

[14] https://neo4j.com/developer/kb/understanding-data-on-disk/.
[15] https://www.imdb.com/interfaces/.

The analysis shows that the number of DBHits was doubled, but the time was only affected by increasing in 11%. We tested if the rules caused significant deterioration in the results with p-paired t test. We found out that no statistically significant differences exist. We also checked if the rules caused improvement or deterioration in each of the sub-groups of queries (the ones that were affected by the change and those which were not affected by the change) and found that in the affected queries there was a significant improvement in both time and DBhits (p-value = 0.029 and 0.043, respectively) and in the un-affected queries the change (either improvement or deterioration) was not significant (p-value = 0.211 and 0.236).

We conclude that applying the rules to the database design improves the performance of the related queries.

5. Design method for the document database and its evaluation

In the following section, we introduce the method for document database model design and its evaluation. The method is demonstrated with the same example as in the previous section, which was introduced in Section 3.

5.1. Fundamentals of document databases

A Document database logical model consists of the following constructs:

- ***Document* and *Collection:*** A document represents an entity in the real world. A collection consists of documents of the same type. A collection has a name that implies its type and potential properties.
- ***Property:*** A document has properties. One or more of the properties of a document is a key, which facilitates its unique identification. A property may be of a complex object type. The values of a property may have constraints, similar to other logical models (e.g., Not-Null, Set).
- ***Embedded Relationship:*** A value of a property may be a document i.e., the document embeds another document.
- ***Referenced Relationship:*** A value of a property may be a reference to a key of another document.
- ***Cardinality constraints*:** Cardinality constraints restrict the number of documents that may participate in a specific type of relationship between documents; the minimum values may be 0 or 1, while the maximum values may range from 1 to many. In the following, we use the same notations from the previous section, with the following additions:
  - $Car\left(c_i, r_i\right)$ = the cardinality of a class c_i in a specific relationship r_i
  - $QS\left(c_i\right)$ = a group of queries in which the c_i class plays the role of a subject. E.g., in query 5 in Listing 1: (RETURN Person.All FROM Watch Item, Person WHERE title = ? and rel = Actor), the subject is Person, as it is the main interest of the query.

5.2. Transformation rules that consider the data requirements

Like the previous case of transforming a class diagram to a Graph database schema, the transformation of a class diagram to a Document database schema is a two-phase process. First, we transform some particular constructs in the class diagram into equivalent binary relationships to enable transforming it into a Document database design. Second, we transform the adjusted class diagram into a Document database schema.

5.2.1. Adjusting the original class diagram

In this phase, we transform the Hierarchy relationships, Aggregation and Composition relationships, and Association classes to equivalent binary relationships:

Hierarchy (inheritance) Relationships

As in the case of Graph databases, handling hierarchy relationships requires observing if there are ***hierarchy constraints*** in the class diagram, which may be Complete (Total cover) or Disjoint/Overlap, and observing the ***queries*** that involve classes in the hierarchy. When dealing with such constraints, two alternatives exist, as discussed in the section on Graph database. While both Graph and Document rules for Hierarchy transformation share similarities, they are not the same, as can be seen in Rule D1:

Rule D1:

1. If *at most* one hierarchy constraint is defined (i.e., only Complete or only Disjoint), remove the sub-classes. We choose this strategy since either there are objects that do not belong to one of the sub-classes, or an object may belong to many sub-classes.
2. If *both* Complete and Disjoint constraints are defined, the relevant queries should be examined:
   a. If there exists a query that references the super-class and no other classes in the "from" part of the query (i.e., no relationship is required in the query), then remove the sub-classes, as in the previous case.
   b. If no such query exists, then remove the super-class. This is because all objects belong to one sub-class only, and there is no query that requires the super-class. Therefore, there is no need to maintain the super-class.

In the IMDb-like application, the hierarchy where *Episode*, *Movie* and *Series* are sub-classes of *WatchItem,* will be transformed based on Rule D1.2.a. Since no other *WatchItem* type exists, and watch items cannot overlap, we conclude that the hierarchy is Complete and Disjoint. Lastly, there exists a query in Listing1 that references the super-class and no other class in the "from" part of the query. Therefore, as rule D1.2.a states, we will remove the sub-classes from the class diagram.

Formal definition: $\exists\left(c_1, \ldots, c_n\right) \prec_i c_{super}$

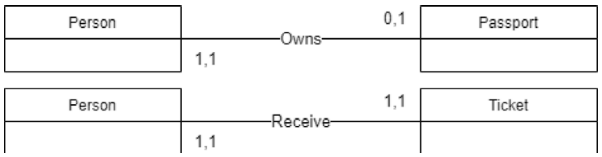


Fig. 7. One-to-One examples.

1. type($\prec_i$) = () or (disjoint) or (complete) $\rightarrow Cl = Cl - (c_1, \ldots, c_n)$ *and* $A(c_{super}) = A(c_{super}) \cup \forall i A(c_i)$ *and* $R = R - \forall i R(c_i, C_j) + R(c_{super}, C_j), C_j = group\ of\ classes\ which\ are\ related\ to\ (c_1, \ldots, c_n)$
2. type($\prec_i$) = (disjoint, complete), and $\exists c_{super} \in q_i \rightarrow Cl = Cl - (c_1, \ldots, c_n)$ *and* $A(c_{super}) = A(c_{super}) + A((c_1, \ldots, c_n))$ *and* $R = R - R((c_1, \ldots, c_n), C_j) + R(c_{super}, C_j), C_j = group\ of\ classes\ which\ are\ related\ to\ (c_1, \ldots, c_n)$
3. type($\prec_i$) = (disjoint, complete), and $\nexists c_{super} \in q_i \rightarrow Cl = Cl - (c_{super})$ *and* $A((c_1, \ldots, c_n)) = A(c_{super}) + A((c_1, \ldots, c_n))$ *and* $R = R - R(c_{super}, C_j) + R((c_1, \ldots, c_n), C_j), C_j = group\ of\ classes\ which\ are\ related\ to\ c_{super}$

Aggregation and Composition Relationships

The mapping of aggregation and composition relationships is done exactly as in the case of the Graph database design. Thus, no new rule formulation is required.

Association classes

As opposed to a graph database, in which relationships are first-class citizens, in document databases we need to emphasize the relationships. Thus, an association class is mapped to an "ordinary" class with two composite relationships to each of the involved classes, and no new rule formulation is required.

5.3. Transforming the adjusted class diagram to a document databases schema

At this stage, we obtained an adjusted class diagram that consists solely of classes and binary relationships. This diagram can easily be mapped to the Document database schema, where generally, each class may be mapped to a collection of documents or be embedded within another collection (that is, become an embedded property of a document) – depending on the types of relationships and their cardinalities. Hence, the main issue is deciding when a relationship between classes is mapped to reference properties of the documents defined from the involved classes or embedded documents. It all depends on the relationship types and their cardinalities. In the formal definitions, we will demonstrate cases of embedding only since reference relationships are simple to transform as they only require the addition of a reference attribute on both sides of the relationship.

5.3.1. Mapping one-to-one relationships

According to Imam et al. [5], based on experiments that they have performed, the authors concluded that one-to-one relationships have better performance when transformed to **embedded** properties of the involved documents. However, they did not discuss how and if such embedding is possible. A description of the possible embeddings of one-to-one relationships can be found in the concept lattices of Varga et al. [14]. These lattices describe possible embeddings based on the *minimum* cardinality, whether it is 0 or 1. Based on that, we distinguish between the following possible cases — see examples in Fig. 7:

a. In case the minimum cardinality of a class is 0, we cannot embed another class within it, as it would result in data loss. Therefore, in cases when both ends of the relationship is 0, it is not possible to embed the relationship. In cases when we only have one end with a minimum 0, we can embed the 0- sided class within the 1-end. For example, in Fig. 7, a *Person* may have a *Passport*, while a *Passport* must belong to one *Person*. If we will embed the *Person* class as a property of the *Passport* document, the data about all persons who do not have a passport will be lost.
b. In case the minimum constraint of a class is 1 we can embed another class within it. In cases when the minimum of both sides of the relationship is 1, embedding is possible yet requires further analysis. For example, in the relationships between Person and Ticket (the second example in Fig. 7), we must choose the best possible embedding between the two possible options. This can be based on the **read** queries. In the said case, we choose to embed the **less frequently** queried class into the **more frequently** queried class to enable getting all information in one read. In this example, we assume that we need to read more frequently the *Person* document (such as the room details, payment details, ticket details, etc.), as opposed to ticket details; this leads to embedding the *Ticket* class within the *Person* document.

Based on the above, we define the following rule:

Rule D2:

In One-to-One relationships, we observe the **minimum** cardinality constraints:

a. If in both classes the minimum is 0, then embedding is **impossible**, and therefore a two-way reference property should be defined.

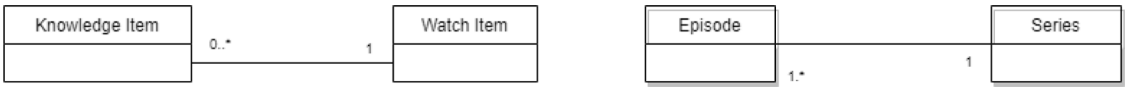


Fig. 8. One-to-Many examples.

b. If one of the classes has a minimum 0, while the other has a minimum 1, then we embed the "may be" document (where the minimum is 0) within the "must be" document (where the minimum is 1).
c. If in both classes the minimum is 1, embedding is possible in both directions. In such cases, we analyze the frequency of the reads queries of the two classes and embed the less frequently read class within the more frequently read class.

Formal definition: $c_1, c_2 \in r_i$

1. $Car\left(c_1, r_1\right) = (0,1)\ and\ Car\left(c_2, r_1\right) = (1,1) \rightarrow Cl = Cl - \left(c_1\right)\ and\ A\left(c_2\right) = A\left(c_2\right) + c_1$
2. $Car\left(c_1, r_1\right) = (1,1)\ and\ Car\left(c_2, r_1\right) = (1,1)\ and\ \#Q\left(c_1\right) < \#Q(c_2) \rightarrow Cl = Cl - \left(c_1\right)\ and\ A\left(c_2\right) = A\left(c_2\right) + c_1$
3. $Car\left(c_1, r_1\right) = (1,1)\ and\ Car\left(c_2, r_1\right) = (1,1)\ and\ \#Q\left(c_1\right) > \#Q(c_2) \rightarrow Cl = Cl - \left(c_2\right)\ and\ A\left(c_1\right) = A\left(c_1\right) + c_2$

5.3.2. Mapping one-to-many relationship

In the case of one-to-many relationship, Imam et al. [5] distinguished between three types of "many": Few (up to 7 documents, denoted F), Many (up to 5000 documents, denoted M), and Squillion (more than 500,000 documents, denoted S). We focus on Few and Many since Squillion may appear in rare cases. In their experiments, 1-to-F relationships performed better when embedding, while 1-to-M relationships performed better when referencing. The authors distinguished between read and write operations. In a read operation, the difference in performance between embedding and referencing was small and not stated if statistically significant.

Due to these experiments and the MongoDB recommendation[16] that suggests that we should reduce the number of operations and get all required data in one read, we recommend embedding one-to-many relationships as well. Like the case of one-to-one relationship, embedding is possible only when the minimum cardinality of the class is 1 (and not 0) to avoid data loss. Herrero et al. [17] suggest that in such cases the class at the 1-side of the relationship be embedded within the document of the many-side of the relationship. This is probably due to possible data duplication, in which the one-side will be embedded in each of the documents of the many-side. We defined this in previous work [21] as multi-embedding, i.e., the embedding of a class in different classes and contexts. We claim that in such relationships embedding is possible in both directions.

In Fig. 8, we present examples of two one-to-many relationships (taken from the class diagram in Fig. 1). In both examples, we prefer to embed the class at the many-side within the class on the 1-side of the relationship — due to the possible queries. In our example, we will probably have more queries regarding a *WatchItem* and its related knowledge items, aka quotes, trivia-items, etc., and fewer queries regarding a *KnowledgeItem*. Thus, we will embed *KnowledgeItem* within *WatchItem*. This is also the case with *Series* and *Episodes*. But reverse embedding is also possible, as shown in the following rule.

Rule D3:

For each class we define 3 counters:

i. how many times a class appears as the subject of read queries - ($Counter_{subject}$)
ii. how many queries exist for both classes together - ($Counter_{joint-queries}$)
iii. how many times the class appears as the subject of queries **when queried together -** ($Counter_{subject-in-joint-queries}$).

1. In case when two classes are queried together (i.e., $Counter_{joint-queries} > 0$), embedding will occur based on $Counter_{subject-in-joint-queries}$ of both classes, in the following manner: we embed the class with the smaller $Counter_{subject-in-joint-queries}$, i.e., appears fewer times as **the subject when queried together** (we will name the said class c_{small}) into the class with the larger counter, i.e., appears more times as **the subject when queried together** (we will name the said class c_{large}), if possible and the minimum cardinality is 1.
   However, we will also check $Counter_{subject}$ - the number of queries in which c_{small} is **generally a subject of query**. If $Counter_{subject}$ for c_{small} is *significantly* larger than $Counter_{joint-queries}$ (in other words, if c_{small} is the subject of many queries), we will not embed the said relationship and create a reference relationship, as c_{small} is required as subject for many other queries.
2. If not, embed based on $Counter_{subject}$: embed the class with the smaller $Counter_{subject}$ into the class with the larger $Counter_{subject}$, if possible (i.e., minimum cardinality is 1).
3. If embedding was not performed, create a two-way reference relationship.

Formal definition: $c_1, c_2 \in r_i . \exists Q\left(c_1, c_2\right) = G, let G_{c1} = QS\left(c_1\right) \subseteq G\ and\ let\ G_{c2} = QS\left(c_2\right) \subseteq G.$

1. $G_{c1} < G_{c2}\ and\ QS\left(c_1\right) \leq QS\left(c_2\right)\ and\ Car\left(c_2, r_{c_1,c_2}\right) = (1,n) \rightarrow Cl = Cl - \left(c_1\right)\ and\ A\left(c_2\right) = A\left(c_2\right) + c_1$
2. $Q\left(c_1, c_2\right) = 0\ and\ QS\left(c_1\right) < QS\left(c_2\right)\ and\ Car\left(c_2, r_{c_1,c_2}\right) = (1,n) \rightarrow Cl = Cl - \left(c_1\right)\ and\ A\left(c_2\right) = A\left(c_2\right) + c_1$

[16] https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/.

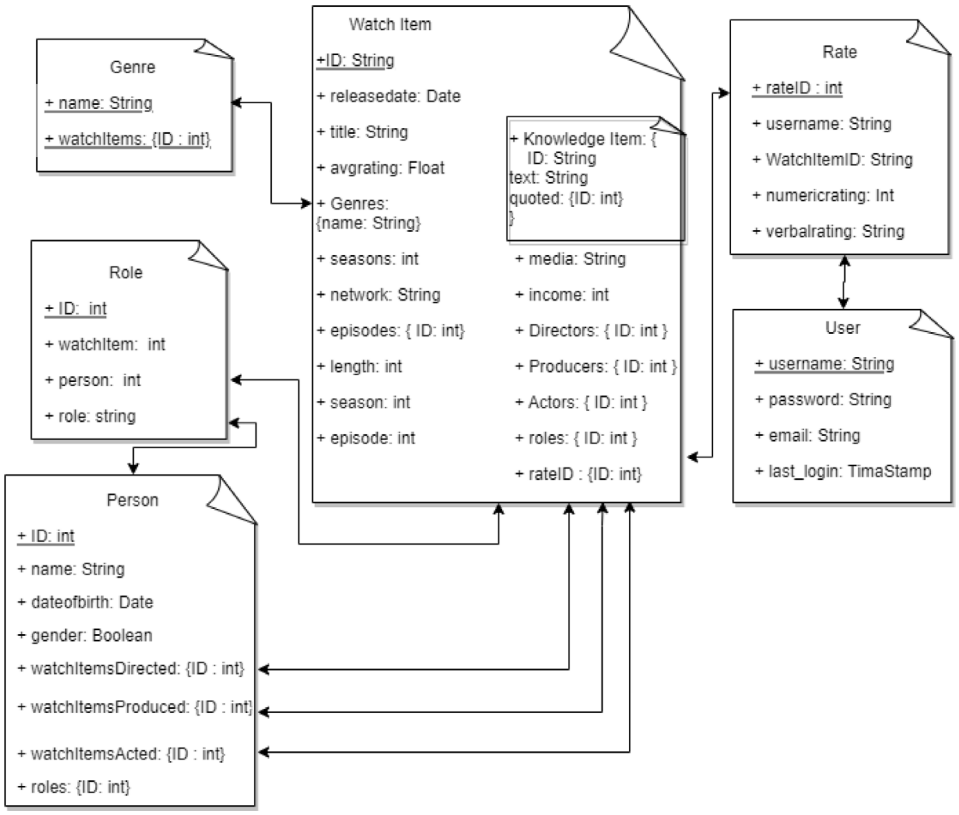


Fig. 9. Document database diagram for the IMDb application after applying the transformation rules.

5.3.3. Mapping many-to-many relationship

Based on their experiments, Herrero et al. [17] suggest not to embed many-to-many relationships. The results of their experiments clearly showed that both writing and reading times are largely reduced when many-to-many relationships are defined as reference relationships. Due to these results, we do not recommend embedding such relationships but rather to define two-way reference properties.

5.3.4. Mapping association classes

Recall that an association class is a class that refers to a relationship between two classes. Therefore, we need to check the relationship based on the cardinalities and convert it accordingly. In case that one participating class is defined to be embedded within the other participating class, then the association class will also be embedded in the same manner. In case embedding cannot be defined between the classes that participate in the association, we will create a new document collection for the said association class and create two-way reference relationships to the adjacent classes. In case a direct relationship is required between the classes, we may create one as well.

Rule D4:

We observe the relationship on which the association class is defined:

1. If the relationship is transformed into a reference relationship in one of the previous steps, a new class will be created to represent the association class. Its content will be a unique ID, the defined properties, and a reference to the keys of the adjacent classes. In the adjacent classes, a reference to the new association class will be created.
2. If the relationship is transformed into an embedded relationship in one of the previous steps, this class will be embedded in the same manner, in the same embedded class.

Formal representation: $letr_i(c_1, c_2), c_3\ association\ class\ of\ r_i$

1. $if\ c_1\ embedded\ in\ c_2 \rightarrow Cl = Cl - (c_3)\ \ and\ A(c_2) = A(c_2) + c_3$
2. $else \rightarrow Cl = Cl + c_3, A(c_3) = A(c_3) + id, A(c_2), A(c_3) = A(c_2), A(c_3) + id$

Fig. 9 demonstrates the recommended Document database schema of the IMDb application. The document shapes denote a collection of documents with all the document properties. The arrows are included for readability purposes only and denote the

reference relationships, **in addition** to the properties added due to the relationships. Examples of such properties are Actors, Directors, and Producers, which are properties within the *WatchItem* document. In a *WatchItem* document, we also embed the *KnowledgeItem* –which does not exist outside the scope of a *WatchItem* as a collection of its own.

5.4. Further transformation rules considering the functional requirements

As in the Graph database design method, consideration of functional requirements may result in transformation rules that will improve the Document database schema obtained. Thus, we will use the same rules that were defined in the previous section:

- Adding generic relationships in addition to several specific relationships between documents.
- Adding relationships that address sequences between items that construct a sequence; and
- Adding implicit relationship.

As described in the previous section, the rules will be applied based on the queries defined in the requirements elicitation phase. The application of these rules is as discussed in the section on Graph database design, with the adaptation to Document databases determining the relationship type will be based on the added relationships' cardinalities, with the relevant cardinalities' rules that were defined previously.

5.5. Evaluation of the document database designed with considering functional requirements

In the previous case, we evaluated the performance of Graph databases designed with the consideration of functional requirements compared to databases designed without them. In the experimental evaluation of the Document databases designed, we followed the similar procedure and the same applications as before. In the following, we describe the experiments and their results. All relevant materials are available in an open Dropbox folder.[17]

5.5.1. Experiment hypothesis and variables and technical set-up.

As in the previous experiments, we hypothesize that when applying the transformation rules that we defined and considering the data and functional requirements, the performance of the Document database will improve compared to a simple Document database with reference relationships **only** as a baseline. Since, in the case of document databases, we wish to check the impact of the relationship type suggested by the method, we created two databases, identical in their relationships' content while differing in the relationships' types, and did not create one database for each functional rule. Therefore, we set the following statistical hypothesis to be tested:

$$H_0 : Performance_{BL}^{Factor} = Performance_{Method}^{Factor}$$

BL stands for baseline system, i.e., with reference relationships only, Method refers to the proposed set of rules, and factor refers to the various measures.

The evaluation was carried out using a MongoDB Document database (MongoDB Compass 1.28.1), as it is the leading Document database provider nowadays, installed over a virtual machine with 8 GB RAM, 250 GB disk size, and Windows-OS. We measured the query execution time, and the number of query stages: in MongoDB, complex queries are written as a pipeline that contains different stages, which are executed one at a time. Stages increase the query writing complexity and the execution time. As MongoDB does not provide information regarding DB hits, we did not include this measure in this experiment.

5.5.2. Evaluation of the IMDb application

For the IMDb application, we used the same data we used for the Graph database experiment. We transformed the CSV files into JSON files using Python code, one for each collection needed either by the database created by applying the method, or by what we defined as the baseline. Table 9 presents the data included in the collections of documents, for both the baseline and the Method databases. (Again, we use the term BL for the databases created using reference relationships only). As opposed to Graph database experiments, where applying the rules creates more labels (for either nodes or relationships), in this experiment the transformation rules applied with this method created fewer collections since, in this case, some collections that exist in the baseline database were transformed into embedded documents in other collections (for example, *KnowledgeItem* within *WatchItem*). The average size of a document in a collection also varies.

In both databases, we executed the same eight queries from Listing 1. The queries were written in MongoDB, via MongoDB Compass, a UI for MongoDB, either by a simple query (which can be addressed as a pipeline with one stage) or by a pipeline with several stages. After writing a pipeline that returns correct results, we used the "explain" command[18] to get the pipeline's execution statistics.

Results and Analysis

Table 10 presents the results of the experiment with the IMDb application.

The results indicate significant improvements: Applying the method reduces the number of stages in the pipeline by 32% and the execution time by 83%. We examined the significance of the differences by performing paired t-test for both measures. We found

[17] https://wwwwww.dropbox.com/sh/uzu1dnnsj3u6m8d/AABMfxTpBfEaQb6yk5gnHW6ca?dl=0.

[18] https://docs.mongodb.com/manual/reference/explain-results/.

Table 9
Statistics of collection in MongoDB.

Collection	# of documents	Avg. document size
Baseline database		
Genre	21	2 kB
Goofs	7661	320.9 B
KnowledgeItem	38 342	62.6 B
Person	1907	258 B
Quotes	11 913	403.7 B
Roles	10 273	81.9 B
Trivia	18 768	309.1 B
WatchItem	1030	770.2 B
Method database		
Genre	21	2 kB
Person	1907	198.8 B
Roles	10 273	81.9 B
WatchItem	1030	12.6 kB

Table 10
The IMDb application: Time in ms and # stages on the two Document databases.

		Q2	Q3	Q4	Q5	Q6	Q7	Q9	Q10	Sum
Baseline database	Time	31	73	12	2	3	5	3	287	416
	Stages	5	8	6	3	1	8	7	11	49
Method database	Time	7	26	12	2	3	3	4	13	70
	Stages	1	4	6	3	1	4	7	7	33

Table 11
Statistics of collection statistics in MongoDB.

Collection	# of documents	Avg. document size
Baseline database		
Category	8	2.9 kB
Customers	91	361.6 B
Details	2155	96.7 B
Employee	9	1.7 kB
Order	830	419.4 B
Product	77	663.2 B
Region	4	192.0 B
Shipper	3	2.4 kB
Supplier	29	340.2 B
Territory	53	85.9 B
Method database		
Category	8	24.7 kB
Customers	91	361.6 B
Employee	9	1.7 kB
Order	830	561.9 B
Product	77	2.1 kB
Region	4	192.0 B
Shipper	3	2.4 kB
Supplier	29	340.2 B
Territory	53	85.9 B

that the difference in the number of stages in the pipelines is indeed significant (p-value = 0.016). The execution time decreased but is not statistically significant (p-value = 0.118). To assess the effect size of the design over the difference in the number of the query stages, we calculated the Hedges's g for a small number of samples (<20). We found out that the application of the method has a medium effect (0.634), i.e., the difference in execution time is significant and matters.

5.5.3. Evaluation of the Northwind database

The experiment with the Northwind application was carried out in the same way as with the IMDb application. The difference in the number of collections between the baseline and the method databases is smaller, as shown in Table 11. In both databases, all ten queries of Listing 2 were defined in MongoDB Compass.

Results and Analysis

As in the IMDb application, we see an improvement in both measures, even though the differences in the number of collections are smaller (see Table 12). However, in this application, we utilized the multi-embedding feature, a concept we explained in previous

Table 12
The Northwind application: Time in ms and # stages on the two Document databases.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Sum
Baseline database	Time	189	4	183	≪1	≪1	8	59	≪1	3	5	35
	Stages	5	1	7	1	1	5	7	2	4	2	31
Method database	Time	14	≪1	26	≪1	≪1	11	37	≪1	4	3	95
	Stages	4	1	6	1	1	4	6	2	4	2	451

Table 13
Summary of the "big-data" IMDb document databases.

Collection	# of documents	Avg. document size
Baseline database		
Genre	28	740.73 kB
KnowledgeItem	101K	86.00 B
Person	1M	121.00 B
Quotes	101K	72.00 B
Roles	1M	88.00 B
WatchItem	1.8M	159.00 B
Method database		
Genre	28	740.73 kB
Person	1M	124.00 B
Roles	1M	88.00 B
WatchItem	1.8M	160.00 B

work [21]. Multi-embedding describes a document that is embedded in several documents, possibly in different contexts, as in the case of the "Order-details" class, which was embedded in different collections based on the functional requirements. This assisted in achieving improvements for both the time of execution and the number of stages. Performing a paired t-test indicates that the difference in the number of stages in the pipeline is significant (p-value = 0.018), with small effect ($g = 0.161$). Yet, the time difference is only close to being statistically significant (p-value = 0.068).

5.5.4. Discussion

First, as mentioned in the Graph database evaluation, it is crucial to stress that in both applications we worked with a small set of queries; we may assume that with more queries, the more significant the impact would be.

Second, while in both applications the time of execution did not introduces statistically significant improvements, we can see that individual changes are indeed statistically significant. For example, the execution time of Query 1 in the Northwind database decreased from 189 ms to 14 ms after applying the rules.

Lastly, as the difference in the number of stages of the pipelines is statistically significant, we may conclude that the queries are simpler and easier to write. This might be reflected in the time that it may take a user to write the queries, which was not measured in this evaluation, yet is an important factor.

5.6. Document database scalability analysis

As in the graph database scalability analysis, we used IMDb, public datasets.[19] The CSV files were transformed into JSON files and then loaded to MongoDB with the same settings as in the experiment in Section 5.5.

We created two databases: the original and one using only referenced relationships. The characteristics of the two databases appear in Table 13.

Again, on the two large databases we execute the relevant queries either as simple query, or as a pipeline (as discussed in Section 5.5.2) and check the results for the significance of the improvements by performing paired t-tests for the number of pipeline stages, and time of execution in milliseconds. Tables 13 14 presents the results in the same way shown in Section 4.4.2. In this case, after further analysis, we noticed that another query would benefit from rule application — query 9 in which we used the generic relationship.

Analysis on the data shows that both time and number of stages are improved with respect to the baseline database. We evaluated these improvements for statical significance with paired t-test. As in the case of smaller IMDb database evaluation in Section 5.5.2, we found that the difference in the number of stages in the pipelines is indeed significant (p-value = 0.016). In contrast to the evaluation of the smaller database, in the large database the p-value for the time measure was very close to being below the threshold for statical significance, with it being = 0.058. This evaluation further confirms our hypothesis, as even though the data was significantly larger than the evaluation in Section 5.5.2, the difference in querying time grew and got close to statistical significance.

[19] https://www.imdb.com/interfaces/.

Table 14
The IMDb application: DB hits and execution time (in ms) for different queries and document databases.

		Q2	Q3	Q4	Q5	Q6	Q7	Q9	Q10	Sum
Baseline database	Time	12 431	15 941	14 408	5157	2000	22 921	7514	14 046	94 418
	Stages	5	8	7	3	1	8	7	11	50
Method database	Time	2554	12 100	18 665	5060	1622	10 449	7214	8590	66 254
	Stages	1	4	7	3	1	4	7	7	34

6. Threats to validity

The results of the experiments should be taken with caution, considering the following threats to validity.

- Small number of queries: The experiments included only eight queries for the IMDb application, and ten queries for the Northwind application. In the case of Graph database design and the IMDb application, only half of the selected queries turned out to be relevant to the transformation rules. More queries that are relevant to the rules may be needed to increase the validity of the results. However, the experiment's results facilitate an understanding of the importance of considering queries in the design process and their potential impact on the database performance, both in Graph and Document databases. In the future, further evaluations using more queries may strengthen the validity of our results.
- Specific DBMSs used: We performed the experiments with one Graph database - Neo4j, and one Document database - MongoDB. However, this seems to be a minor limitation as Neo4j is the leading Graph database provider, and MongoDB is the leading Document database provider, according to DB-engines ranking.
- Data size: While the IMDb application includes a relatively large amount of data, current applications may store a large amount of data, which may impact the effects of the proposed rules. However, since the rules significantly improved the performance of the database, we may assume that it will be even more significant with more data.
- User-based evaluation: while our evaluations examined database performance, it might be important to also examine the users' opinions of the proposed methods. By users, we mean database designers who will apply the methods. It is important to examine how easy it is to learn the methods and apply the transformation rules.

7. Summary

Designing NoSQL databases consists of two stages. The first stage allows the selection of the proper database technologies for a certain application based on the data, functional, and non-functional requirements [22]. The second stage refers to the design of a certain NoSQL database, based on the same set of requirements. In this paper, we refer to the second stage and present two methods for designing database schemas of two NoSQL database management systems -Graph databases and Documents databases. We based the design methods on the users' requirements of the sought systems: data-related requirements, which are expressed as UML class diagrams, and functional requirements, which are expressed as queries. Both methods receive the same inputs, and each method consists of the following steps: first, the initial class diagram is changed/adjusted so that certain of its constructs are mapped to equivalent constructs to enable easier mapping of the class diagram to the target database schema. Then, the adjusted class diagram is mapped to a schema of the target NoSQL database, using certain transformation rules. In the next step, the functional requirements, namely the queries that will be used by the users of the system to retrieve or update the databases, are considered. According to the types of the queries and their frequency of use, certain parts of the target database schemas may be changed to create database schemas that are more efficient in terms of execution time. We have expressed the various rules of transformation formally and demonstrated them using examples taken from a certain application. Then we conducted evaluations of the database schemas that consider the functional requirements compared to the database schemas that do not consider these requirements. The results of the evaluations demonstrated significant improvement: Graph databases that were designed with the consideration of functional requirements significantly outperformed the database that was designed without these considerations. In Document databases that were designed according to the proposed rules, the queries were significantly less complex, and their execution times were also improved.

In further evaluations, we plan to include more queries and types of queries to further validate the proposed transformation rules. While the rules are agnostic to types of queries, we intend to examine whether the type of the query (i.e., select, insert, update, delete) affects the performance. For example, Mior et al. [18] also referred to update and insert queries in the design process and evaluated the design impact for them as well. Other types of experiments can be found in [23–25] which compare databases with respect to different workloads (i.e., write workload, read workload, and combined). We also plan to conduct user evaluations to find out how easy it is to learn and apply the design methods, and how easy it is to formulate and write queries for database schema designed according to the proposed methods.

Finally, our ongoing work includes implementing both design methods as a single application.[20] This application will enable users to apply the methods without the need to be familiar with the theories and rules behind them. We believe that such application will foster wide adoption of the methods, especially in small software development organizations, e.g., start-ups. Such application will also support further evaluations of the design methods.

[20] https://rps.ise.bgu.ac.il/njsw27.

CRediT authorship contribution statement

Noa Roy-Hubara: Conceptualization, Investigation, Writing. **Arnon Sturm:** Conceptualization, Methodology, Writing – review & editing, Supervision. **Peretz Shoval:** Conceptualization, Methodology, Writing – review & editing, Supervision.

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] N. Roy-Hubara, A. Sturm, Design methods for the new database era: a systematic literature review, Softw. Syst. Model. (2019) 1–16.
[2] F. Abdelhedi, A.A. Brahim, F. Atigui, G. Zurfluh, UmLtoNoSQL: Automatic transformation of conceptual schema to NoSQL databases, in: 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications, AICCSA, IEEE, 2017, pp. 272–279.
[3] R. De Virgilio, A. Maccioni, R. Torlone, Model-driven design of graph databases, in: International Conference on Conceptual Modeling, Springer, 2014, pp. 172–185.
[4] N. Roy-Hubara, L. Rokach, B. Shapira, P. Shoval, Modeling graph database schema, IT Prof. 19 (6) (2017) 34–43, November/2017.
[5] A.A. Imam, S. Basri, R. Ahmad, N. Aziz, M.T. González-Aparicio, New cardinality notations and styles for modeling NoSQL document-store databases, in: TENCON 2017-2017 IEEE Region 10 Conference, IEEE, 2017, pp. 2765–2770.
[6] K. Shin, C. Hwang, H. Jung, NoSQL database design using UML conceptual data model based on Peter Chen's framework, Int. J. Appl. Eng. Res. 12 (5) (2017) 632–636.
[7] H. Huang, Z. Dong, Research on architecture and query performance based on distributed graph database neo4j, in: 2013 3rd International Conference on Consumer Electronics, Communications and Networks, IEEE, 2013, pp. 533–536.
[8] S. Bordoloi, B. Kalita, Designing graph database models from existing relational databases, Int. J. Comput. Appl. 74 (1) (2013).
[9] G. Daniel, G. Sunyé, J. Cabot, UMLtoGraphDB: Mapping conceptual schemas to graph databases, in: Conceptual Modeling: 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Springer, 2016, pp. 430–444.
[10] J. Pokorný, Conceptual and database modelling of graph databases, in: Proceedings of the 20th International Database Engineering & Applications Symposium, 2016, pp. 370–377.
[11] J. Akoka, I. Comyn-Wattiau, N. Prat, A Four V's design approach of NoSQL graph databases, in: International Conference on Conceptual Modeling, Springer, 2017, pp. 58–68.
[12] V.M. de Sousa, L.M.D.V. Cura, Logical design of graph databases from an entity-relationship conceptual model, in: The 20th International Conference on Information Integration and Web-Based Applications & Services, 2018, pp. 183–189.
[13] A. Ghrab, O. Romero, S. Skhiri, A. Vaisman, E. Zimányi, Grad: On graph database modeling, 2016, arXiv preprint arXiv:1602.00503.
[14] V. Varga, K.T. Jánosi-Rancz, B. Kálmán, Conceptual design of document NoSQL database with formal concept analysis, Acta Polytech. Hung. 13 (2) (2016) 229–248.
[15] V. Varga, C.F. Andor, C. Săcărea, Conceptual graphs based modeling of MongoDB data structure and query, in: International Conference on Conceptual Structures, Springer, Cham, 2019, pp. 262–270.
[16] C. Lima, R.S. Mello, On proposing and evaluating a NoSQL document database logical approach, Int. J. Web Inf. Syst. (2016).
[17] V. Herrero, A. Abelló, O. Romero, NOSQL design for analytical workloads: variability matters, in: International Conference on Conceptual Modeling, Springer, Cham, 2016, pp. 50–64.
[18] 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.
[19] S. Scherzinger, S. Sidortschuck, An empirical study on the design and evolution of NoSQL database schemas, in: G. Dobbie, U. Frank, G. Kappel, S.W. Liddle, H.C. Mayr (Eds.), Conceptual Modeling. ER 2020, in: Lecture Notes in Computer Science, vol. 12400, Springer, Cham, 2020, pp. 441–455.
[20] I. Robinson, J. Webber, E. Eifrem, Graph Databases: New Opportunities for Connected Data, O'Reilly Media, Inc, 2015.
[21] N. Roy-Hubara, A. Sturm, P. Shoval, Designing document databases: A comprehensive requirements perspective, in: International Conference on Conceptual Modeling, Springer, Cham, 2021, pp. 15–25.
[22] N. Roy-Hubara, P. Shoval, A. Sturm, Selecting databases for Polyglot Persistence applications, Data Knowl. Eng. 137 (2022) 2022.
[23] P. Martins, M. Abbasi, F. Sá, A study over NoSQL performance, in: World Conference on Information Systems and Technologies, Springer, Cham, 2019, pp. 603–611.
[24] E. Tang, Y. Fan, Performance comparison between five NoSQL databases, in: 2016 7th International Conference on Cloud Computing and Big Data, CCBD, IEEE, 2016, pp. 105–109.
[25] B.G. Tudorica, C. Bucur, A comparison between several NoSQL databases with comments and notes, in: 2011 RoEduNet International Conference 10th Edition: Networking in Education and Research, IEEE, 2011, pp. 1–5.

Noa Roy-Hubara is a Ph.D. student at the Department of Software and Information Systems Engineering at Ben-Gurion University of the Negev, Israel. Her research interests include databases, data modeling, and big data. Roy-Hubara received both her M.Sc. and B.Sc. in Information Systems Engineering at Ben-Gurion University.

Arnon Sturm is a faculty at the Department of Software and Information System Engineering at Ben-Gurion University. His research interests include software engineering, in particular, model-based software development and empirical software engineering, and knowledge management.

Peretz Shoval is a Professor at the Department of Software and Information Systems Engineering at Ben-Gurion University and the Academic College of Netanya. He earned his Ph.D. in Information Systems from the University of Pittsburgh (1981), where he specialized in expert systems for information retrieval. Shoval's research interests include conceptual database modeling, information systems analysis and design methods, and information retrieval & filtering. He has published numerous papers in journals, edited books and conference proceedings, and authored several textbooks on systems analysis and design. Among other things, he developed the ADISSA and FOOM methodologies for systems analysis and design; and methods and tools for conceptual data modeling, view integration and reverse database engineering.