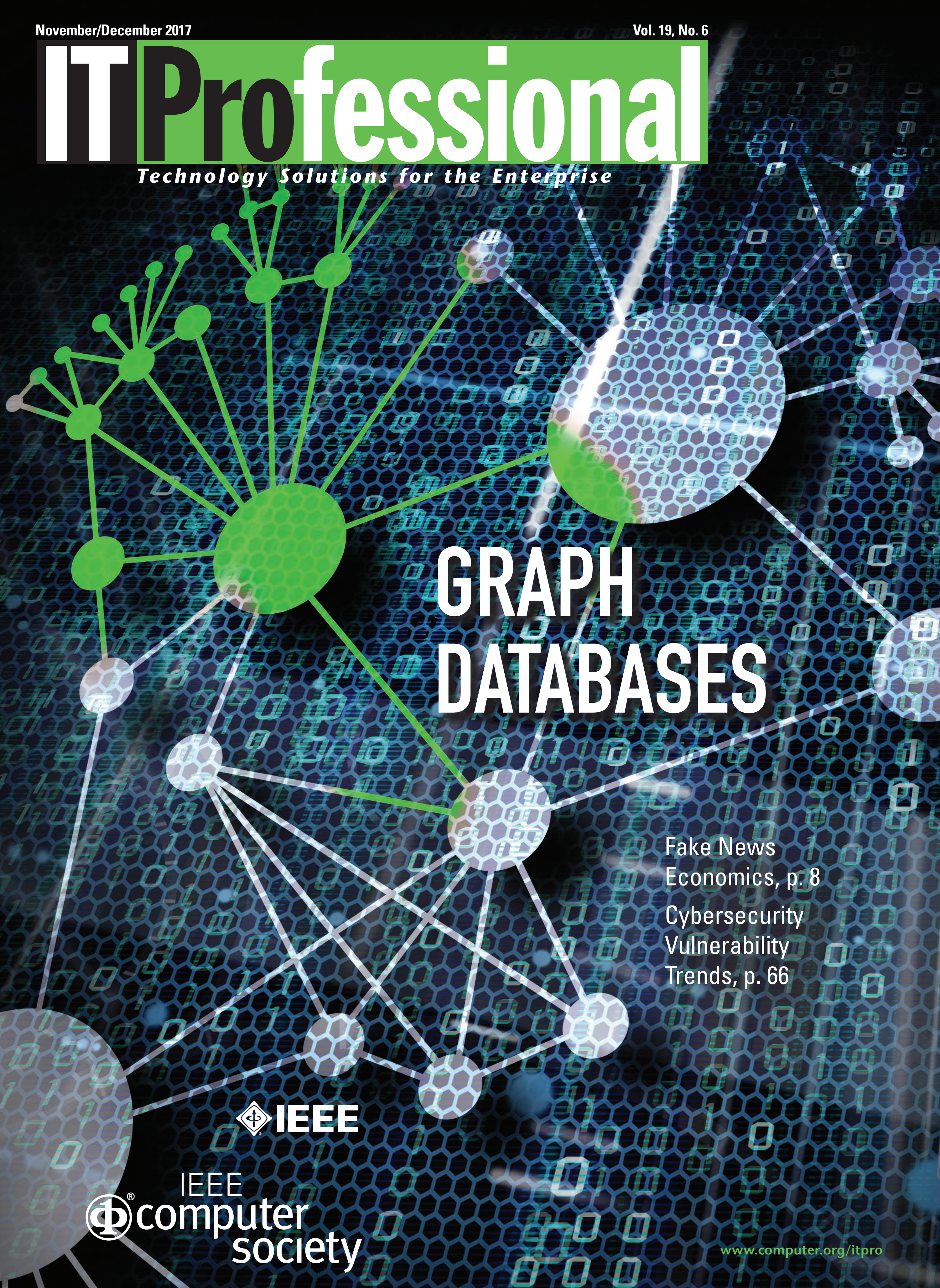
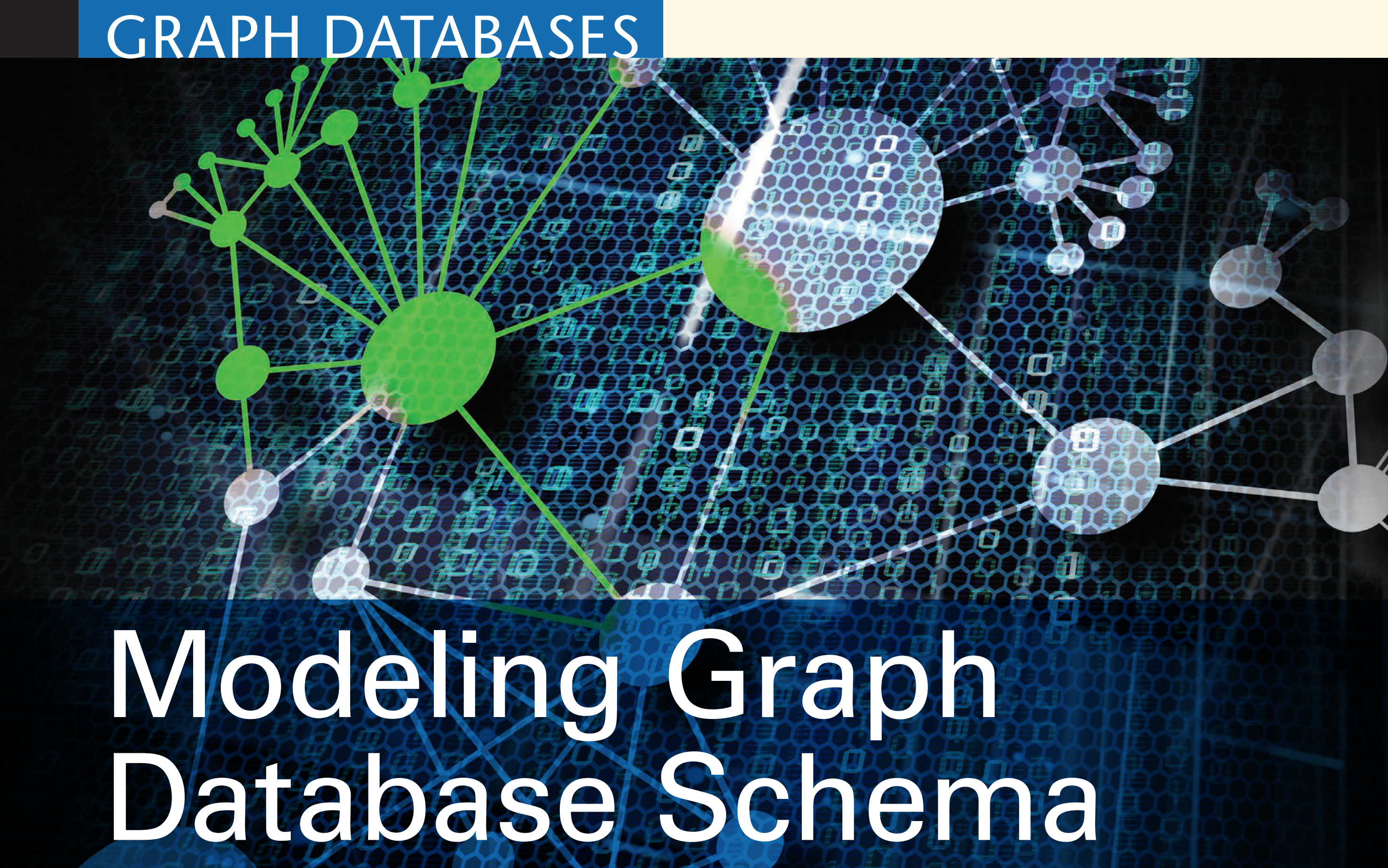

November/December 2017
Vol. 19, No. 6
IT Professional
Technology Solutions for the Enterprise
GRAPH DATABASES
Fake News Economics, p. 8
Cybersecurity Vulnerability Trends, p. 66
IEEE
IEEE computer society
www.computer.org/itpro



Modeling Graph Database Schema

Noa Roy-Hubara, Lior Rokach, and Bracha Shapira, *Ben-Gurion University of the Negev, Israel*
Peretz Shoval, *Ben-Gurion University of the Negev, Israel, and Netanya College*

The authors' approach to creating a graph database schema (GDBS) is based on an entity-relationship diagram of the application domain, which is mapped to a GDBS in a two-step process.

There are many ways to store data. Until recently, data was most commonly stored in relational databases. The evolution of the Web and the explosive growth of big data have placed new demands on database technology, bringing the relational model to its limits. What worked well for many years for structured data is not well suited for the unstructured, massive amounts of data that are part of the Web and new Web applications, such as social networks.

Sam Madden characterizes big data by the three V's:[1] velocity, volume, and variety. Another study lists the characteristics of new demands on database technology: high concurrency of reading and writing with low latency, efficient big data and access requirements, high scalability, high availability, and lower management and operational costs.[2] Relational databases cannot fulfill all these demands, so new types of databases have emerged, including NoSQL ("not only SQL") databases. NoSQL can be categorized into four main types: key-value stores, column-oriented, document stores, and graph databases.[2–4]

The NoSQL world is rapidly developing, but these databases still have many shortcomings that prevent users from adopting them and limit their use. The lack of a database design methodology is one such shortcoming. One claim is that good database design is crucial to obtaining a sound database, and that further research is required for NoSQL to meet the needs of big

 Published by the IEEE Computer Society 1520-9202/17/$33.00 © 2017 IEEE

Related Work in Modeling NoSQL Databases

Several studies have attempted to create a model for NoSQL databases. Some researchers have focused on a specific kind of NoSQL database or on specific providers' databases, whereas others have tried to model all four types of NoSQL databases. Here, we review only two relevant studies, owing to space limitations.

Karamjit Kaur and Rinkle Rani developed a modeling methodology that is suitable for all four types of NoSQL databases.[1] They demonstrated their model using a case study presented with an entity-relationship diagram (ERD). Their case study was then "modeled" on a document-oriented database (MongoDB) and a graph database (Neo4j). They showed how the case study transforms into a database, but the transformation was not backed by rules, explanations, and so on. In our opinion, the article does not provide a methodology but rather demonstrates a use case for document-based and graph NoSQL databases.

Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone proposed a methodology for transforming requirements into a graph database.[2] Their solution is based on an ERD representation of the domain of interest. Their strategy is to aggregate objects based on a weighting function. In essence, the ERD is transformed into an oriented ERD (OERD), a graph based on weighing rules in which each edge receives a numeric value. The authors compared their method to a modeling strategy called Sparse. At first glance, the proposed method seems like a good solution to the modeling problem, but a closer inspection reveals several drawbacks. First, the ERD that they use is limited; it does not contain elements such as weak entities, generalizations, and ternary relations. Second, the weighing function and aggregation rules are not well defined but only explained intuitively; in our opinion, deeper explanations are needed. Finally, the comparison of their method to the alternative is based on query response time, but there are many more nodes in the Sparse modeling strategy compared to the proposed method. Obviously, when considering response time, a method will perform much better if it has to traverse fewer nodes.

Our proposed method for creating a graph database schema overcomes existing limitations. As mentioned in the main text, our method assumes that a conceptual schema of the domain of application exists—a rich ERD that includes various constructs that were not included in previous studies, such as weak entity types, ternary relationships, generalization, and is-a hierarchy. The ERD is then mapped to a graph database schema using specified rules. The resulting schema preserves the integrity constraints of the original conceptual schema.

References

1. K. Kaur and R. Rani, "Modeling and Querying Data in NoSQL Databases," *Proc. IEEE Int'l Conf. Big Data*, 2013; doi:10.1109/BigData.2013.6691765.
2. R. De Virgilio, A. Maccioni, and R. Torlone, "Model-Driven Design of Graph Databases," *Proc. Int'l Conf. Conceptual Modeling*, 2014, pp. 172–185.

data, unstructured data, imperfect data, and the like.[5] Other research claims that the need for logical data models in the NoSQL world is a central issue;[6] it seems, the authors say, that NoSQL systems don't distinguish between logical and physical schemata, which complicates database maintenance. NoSQL databases are considered schema-less, but data modeling and schema will always remain important.[7]

In this study, we focus on graph databases, which work well with interconnected data—that is, data with many relationships.[8,9] These databases provide an easy way to model relationships between different data types, and they have many traversal algorithms that are helpful in finding patterns. The main components of a graph database are nodes and edges (relationships). Edges are usually directed (that is, an edge has a start and an end node). Nodes and edges have labels and might have properties. Graph databases have no schema; some claim that this provides more flexibility.[8,9] However, the flexibility associated with having no schema has drawbacks; without a schema, it is difficult to enforce integrity constraints. For example, a node in the database can be connected to any other node, regardless of whether it makes sense.

Here, we propose a method for modeling graph databases. More specifically, we suggest a method for creating a schema for graph databases based on a conceptual schema of the application domain. We use an entity-relationship diagram

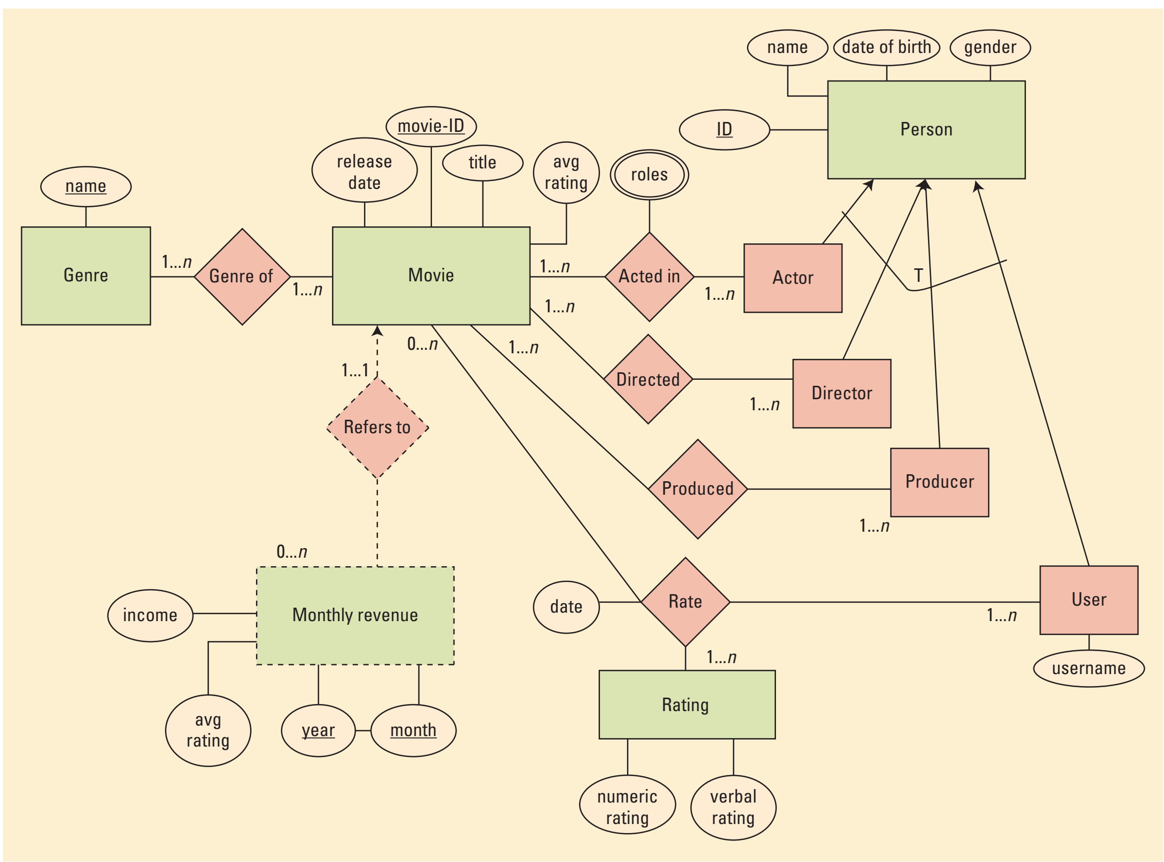


Figure 1. **The entity-relationship diagram for the movie recommendation system. Subtypes are not exclusive—actors can also be directors, producers can be actors, and so on. Also, the subtypes cover all possible people in the system, as indicated by the total cover (T) constraint.**

(ERD) as the conceptual schema and provide a two-step process for mapping the ERD to a graph database schema (GDBS). The proposed method can easily be adapted for use with a class diagram instead of an ERD.

Graph Database Schema

Before presenting our proposed method for creating a graph database schema, we provide an example of the requirements for a graph database in the movie recommendation system domain. We then present the ERD that models these requirements. Later in this section, we define the components of our target GDBS.

Movie Recommendation System Requirements

The example is based on the movie recommendation system domain, which provides information about movies and stores users' movie ratings. The system stores the following information for each movie: a unique ID; name; genre; lists of actors, directors, and producers; release date; average rating; users who rated the movie; and a timestamp for each rating. A user can rate a movie multiple times. The ratings range from 1 to 5 stars.

Each user, director, actor, and producer has a unique ID, name, date of birth, and gender. An actor might act in many movies, a director might direct many movies, and a producer might produce many movies. A movie can be directed by more than one director, produced by more than one producer, and have many actors. An actor can play several roles in the same movie. An actor might also be a director or a producer, and vice versa. In addition, the system stores monthly data for each movie, including monthly revenue and users' average monthly rating.

All this information is stored so that the system can make accurate movie recommendations. Take, for example, a user who watched the movie *Titanic* and gave it a 5-star rating. Based on such information, that user will be able to make the following queries to find other movies that he or she will enjoy:

- Find other users who watched *Titanic* and rated it with 5 stars.
- Find movies directed by the same director as *Titanic* and containing an actor who also acted in *Titanic*.
- Find movies of the same genre as *Titanic* that were released in the same year or later.

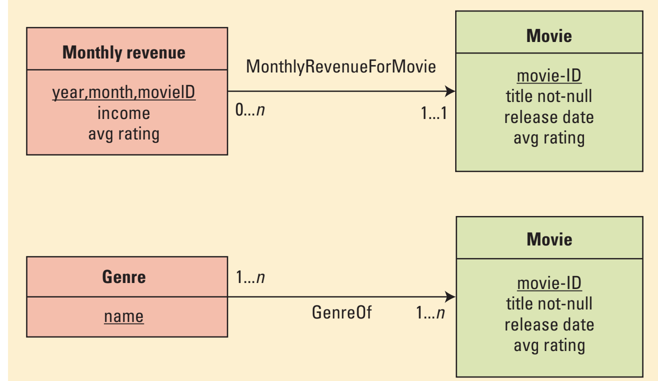


Figure 2. Examples of nodes, edges, properties, and cardinality constraints. A node is represented by a rectangle divided into two parts, the node label (upper) and the property names (lower). An edge is represented by a line with an arrow head pointing from the start node to the end node. Key properties are underlined, and cardinality constraints are denoted using the minimum and maximum number of nodes that can participate in each edge type.

ERD for the Movie Recommendation System

Figure 1 presents the ERD for the aforementioned required system. In the figure, *Actor*, *Director*, *Producer*, and *User* are subtypes of *Person*. These subtypes are not exclusive; an actor can also be a director, a producer can also be an actor, and so on. The subtypes cover all the possible people in the system—that is, there are no other subtypes of *Person*, as indicated by the total cover (T) constraint connecting the subtype arrows.

Rate is a ternary relation (*n*:*n*:*n*) between *User*, *Movie*, and *Rating*: a movie can be rated by many users with different ratings; each rating has a date. *Monthly revenue* is a weak entity that is related to its “strong” entity *Movie*; it stores monthly information for each movie. Note the partial key attributes *year* + *month*.

Graph Database Schema Components

The proposed graph database schema consists of nodes, edges, properties, and cardinality constraints.

Node. A node represents an entity in the real world. Similar to the entity-relationship model, which makes a distinction between entity type and entity occurrence, or to a class diagram, in which there is a distinction between class and object, here a node signifies both a node type and node occurrence. A node has a label (name) and properties, one of which is a key property that enables its unique identification. For example, for the node label *Movie*, the node properties are *movie-ID* (key), *title*, *release date*, and *average* (*avg.*) *rating*. In Figure 2, a node is represented by a rectangle divided into two parts: the upper (smaller) part includes the node label; the lower (larger) part includes the property names. (This is similar to a graphic symbol of a class in a class diagram, except that here the node does not include method names.) Note that the key attribute is underlined.

Edge. An edge represents a relationship between two nodes. Similar to nodes, here, too, we distinguish between edge type and edge occurrence, which is an edge between two specific nodes.

As in most graph databases,[8] an edge has a direction—that is, it has a start node and an end node. An edge might also have properties. An edge has a name (which need not be unique because it can be identified by its start and end nodes).

For example, for the edge name *actors in movies,* the start node is *Movie* and the end node is *Actor*. The property is *roles* (enabling an actor to play more than one role in a movie). We could define the start and end nodes in the opposite direction as well; it actually does not matter, because in either direction, the graph database

system will be able to answer a given request, possibly by posing the query somewhat differently. An edge is represented by a line with an arrow head pointing from the start node to the end node. The edge name is written above or near the arrow.

Property. As said previously, both nodes and edges can have properties. Properties might have constraints:

- *Key*. Each node has a key that might be one or a combination of several properties. For example, *movie-ID* is a key of *Movie*; *movie-ID* + *year* + *month* together are the key for *Monthly revenue*.
- *Not-null*. The property must have a value for all instances of the node or edge. For example, *title* is a not-null property of *Movie* (meaning that every movie must have a title).
- *Set*. The property can have many values. For example, *roles* is a set property of the edge between *Movie* and *Actor*, given that an actor might play many roles in a movie.
- *Index*. The system maintains an index for this property. It can be assumed that the key property implies that it is indexed, but any other property can also be indexed. (The question of which properties to index is beyond this article's scope.)

In Figure 2, properties of a node are written in the bottom part of the node's rectangle, starting with the (underlined) key property. Properties of the edge, if they exist, are written in parentheses after the edge name. A constraint of a property, if it exists, is written after the property name.

Cardinality constraints. The proposed graph database schema, which extends existing graph databases, includes cardinality constraints between the nodes of each edge. For this, we use the same notations used in the ERD—that is, next to each node, we write the minimum and maximum number of nodes that can participate in each edge type.

For example, Figure 2 shows a 1:*n* relationship between *Movie* and *Monthly revenue*, and an *n*:*n* relationship between *Movie* and *Genre*. Note the similarity to the respective relationships in Figure 1.

Mapping an ERD to a Graph Database Schema

The method for creating the aforementioned GDBS from an ERD consists of two steps: adjusting the ERD so it is ready for GDBS creation, and mapping the adjusted ERD to the GDBS.

Adjusting the ERD

An ERD can include constructs that cannot be mapped directly to a GDBS, which consists of only nodes and binary edges. So, in the first step, we adjust some of the original ERD constructs to semantically equivalent ERD constructs that can be mapped later on. Specifically, we adjust the following constructs: ternary relationships, aggregation (whole-parts) relationships, and inheritance (is-a) relationships.

Ternary relationships. A ternary relationship is mapped to a weak entity, with binary relations to the entities involved in the ternary relation. If the relation has properties, they are added to the weak entity. The name of the weak entity might be identical to the name of the original ternary relationship, or be composed of the names of the involved entities, or be any name that resembles its role.

For example, the ternary relation among the entities *User*, *Movie*, and *Rating* is mapped to a weak entity with three binary relationships to these entities, as Figure 3 shows. In this example, the ternary relation is *n*:*n*:*n*; therefore, the weak entity has weak binary relationships with its three strong entities. In cases in which the ternary relation is *n*:*n*:1, the weak entity would have two weak binary relationships with the strong entities, which are on the "*n*" side of the ternary relationship, and an ordinary relationship with the entity on the "1" side.

Aggregation (whole-parts) relationships. An aggregation relationship (for example, a car is composed of an engine, wheels, gears, and so on) is mapped to an ordinary binary relationship in which the cardinality of the "parts" entity is 0:*n* or 1:*n* (depending on whether the parts entity is mandatory), whereas the cardinality of the "whole" entity is always 1:1. (Due to space limitations, an example for this case is not included in this article.)

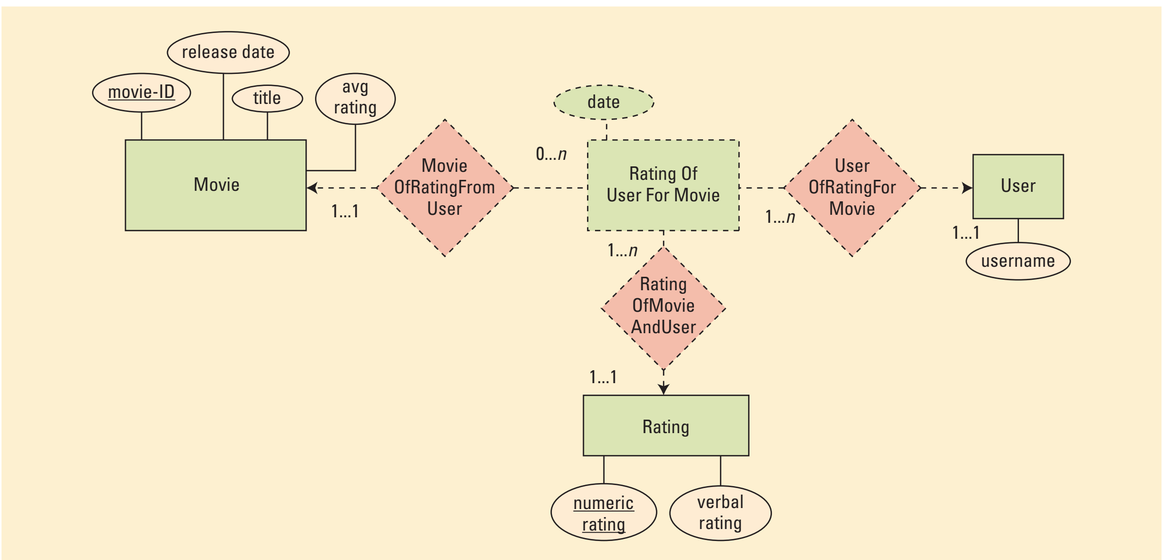


Figure 3. Ternary relation to three binary relationships. The ternary relation is *n*:*n*:*n*, so the weak entity has binary relationships with its three strong entities.

Inheritance (is-a) relationships. Inheritance relationships can be mapped in two different ways. The question of which of the two is better in terms of efficiency and time to process queries will be addressed in further research.

In method A, there is no change of the original entities. According to this method, the involved entities remain unchanged, but the is-a relationships are mapped to ordinary binary relationships, named is-a, with cardinalities 1:1 next to each of the involved entities. For example, if a *Person* (super-type entity) exists, say "Tom Hanks," then there is only one *Actor* (subtype entity) that is that "Tom Hanks."

In method B, we remove the inheritance relationships and merge the super-type with the subtypes. We distinguish between two possibilities. The first is removing the subentities and moving their attributes and relationships to the super-type entity, adding to it a new property named *type*. This mapping is applied when the inheritance relationship is not defined with the T constraint nor with the exclusive (X) constraint, meaning that there are super-types that are not one of the subtypes or that a super-type might belong to many subtypes. If this inheritance relationship does not have an X, then the super-type's new property *type* will be defined as a set to allow it to contain more than one value.

For example, in our ERD, *Person* is the super-type of four subtypes. Assuming that there is no T constraint, and given that there is no X constraint, the four subtypes will be removed, and their properties and relationships will be added to *Person*, as shown in option 1 of Figure 4.

The second possibility is removing the super-entity and moving its properties and relationships to each of its subtypes. This mapping is applied in cases in which there is a T *and* X constraint between the subtypes, meaning that all super-types belong to one subtype only. Therefore, there is no need to maintain the super-type. For example, if in our ERD the inheritance relationship between *Person* and its four subtypes had an X constraint in addition to the T constraint, the adjusted ERD would look like that shown in option 2 of Figure 4.

Mapping the Adjusted ERD to the GDBS

The mapping process consists of the following steps.

Mapping entities to nodes. Each entity is mapped to a node, and the entity's properties become the node's properties. For example, the *Movie* entity is mapped as follows:

- Node label: *Movie*

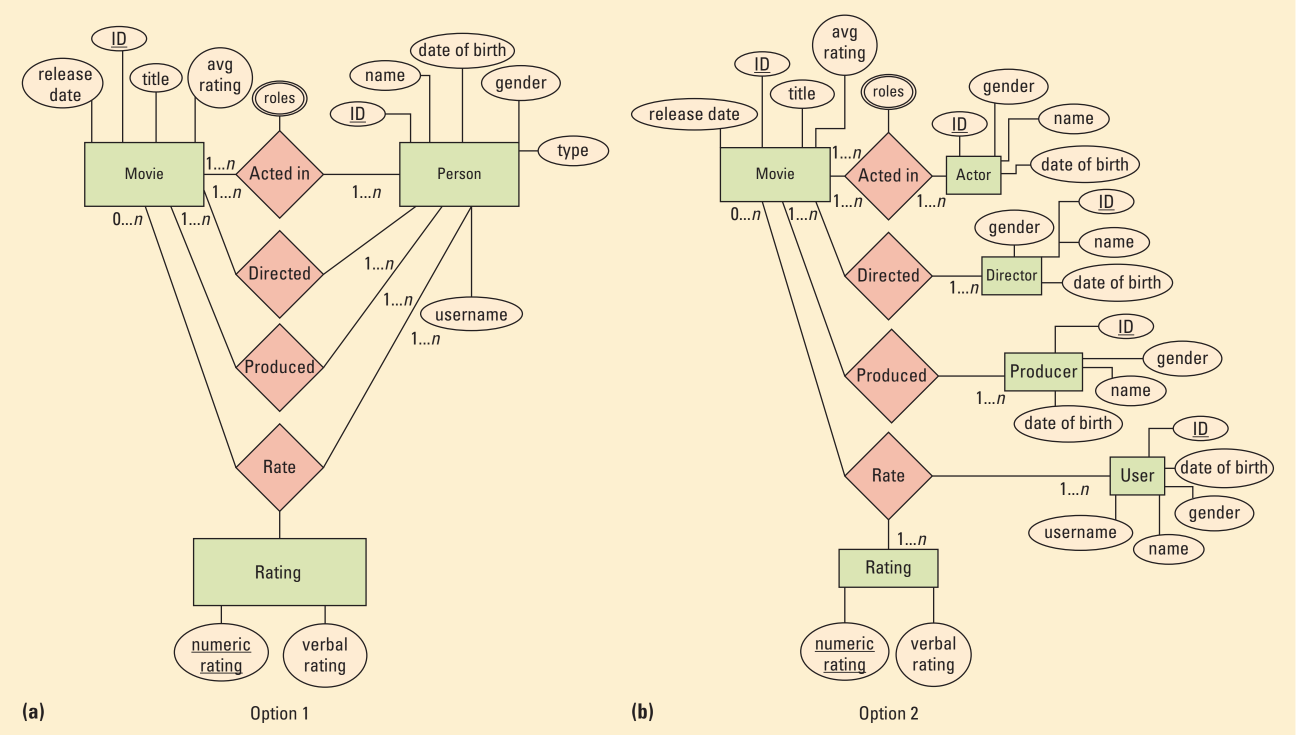


Figure 4. Handling inheritance relationships. (a) In option 1, assuming that there is no T constraint or no X constraint, the four subtypes will be removed, and their properties and relationships will be added to *Person*. (b) In option 2, the inheritance relationship between *Person* and its four subtypes has T and X constraints.

- Properties: *movie-ID* (key), *title* (not null), *release date*, *avg. rating*

A weak entity is mapped to a node just like an ordinary entity. The key property of this node is composed of the keys of the related strong entities, plus the partial key of the weak entity (if this exists). For example, the *Monthly revenue* entity is mapped as follows:

- Node label: *Monthly revenue*
- Properties: (*movie-ID* + *year* + *month*) (key), *income*, *avg. rating*

Mapping relationships to edges. Each relationship between entities is mapped to an edge connecting the respective nodes. The edge's name might be the name of the relationship or be based on the roles of the connected nodes.

Unlike with an ERD, in a graph database, the edges are directed, distinguishing between the start and end nodes of each edge. For example, for the relationship between *Movie* and *Genre*, there are two mapping options:

- *Movie* → *Genre* (that is, the start node is *Movie*; the end node is *Genre*); possible name: "genre of movie"
- *Genre* → *Movie*; possible name: "movies of genre"

It doesn't matter which of the two nodes is defined as the start node and which is defined as the end node, because, as said, the graph database enables traversing from node to node in any direction.[9]

Mapping cardinality constraints. To the best of our knowledge, current graph databases do not define cardinality constraints—that is, the minimum and maximum number of nodes that can participate in an edge type. For example, Neo4j,[8] a leading graph database system, has not (yet) defined such constraints.

We propose adding cardinality constraints to edges, equivalent to such constraints in the

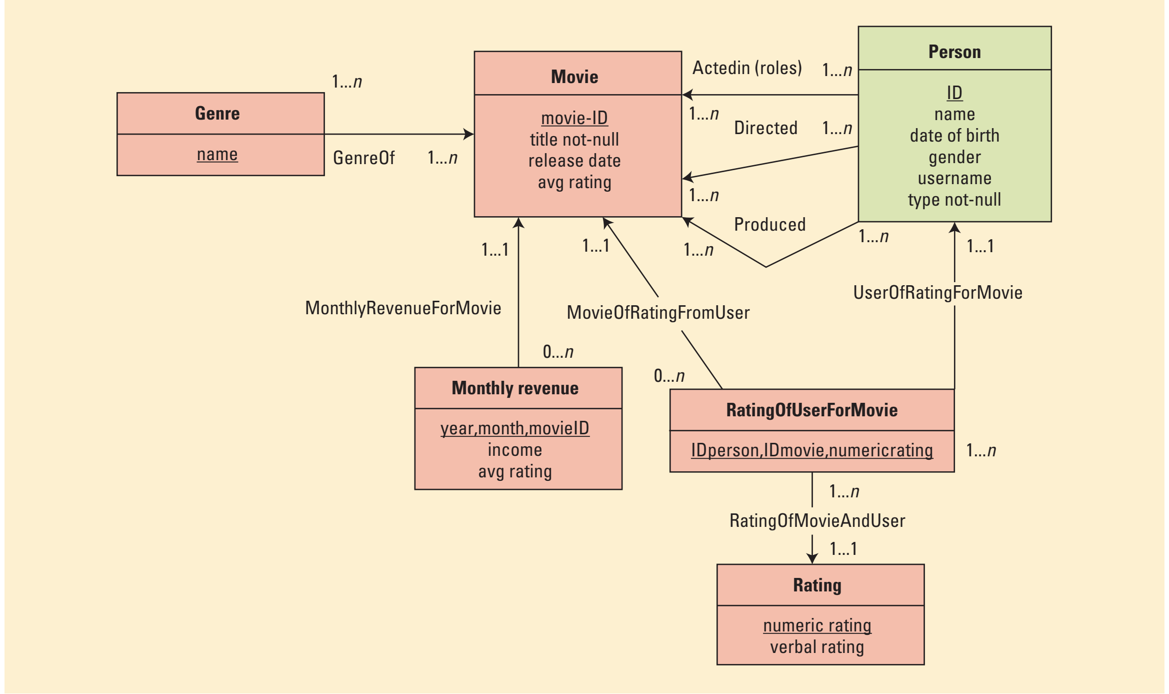


Figure 5. Graph database schema diagram. We show one possible mapping—namely, when inheritance relationships are mapped to remove subtype entities and move their attributes and relationships up to the super-type entity.

entity-relationship model. It is important to include such constraints because in some cases, we want to limit the number of nodes that can be involved in an edge type. Examples of such constraints are as follows: a movie may have no less than 2 and no more than 10 actors; a user may rate a certain movie only once.

The mapping of cardinality constraints from the adjusted ERD to the GDBS is very simple: we adopt the same notations as shown in Figure 5.

The Resulting GDBS

Figure 5 presents the resulting GDBS. Due to space limitations, we show only one of the possible mappings, namely, when the inheritance relationships are mapped so that the subtype entities are removed and their attributes and relationships are moved up to the super-type entity.

Data Definition Language of the GDBS

Based on the resulting GDBS diagram, Figure 6 defines a data definition language of the schema, which can be added to a graph database system such as Neo4j.

In our graph database modeling approach, the ERD is adjusted to an equivalent ERD that can be mapped; then, using specific rules, the adjusted ERD is mapped to a GDBS consisting of nodes, edges, properties, and cardinality constraints—thus preserving the integrity constraint defined in the original ERD.

In future work, we plan to test the proposed method. First, we plan to compare the two possible mappings of an original ERD to an adjusted ERD, as described previously. Second, we plan to compare our method to existing methods of creating graph databases (which are not actually formal methods but practitioners' best practices). These comparisons will involve creating graph databases for one or more domains (possibly using existing databases), and utilizing different graph database schemata and an existing graph database system (such as Neo4j). Then, we will run various queries and measure the execution time. In addition, we plan to conduct controlled experiments (possibly with students) to measure and compare the ease of use and quality of graph database schemas created with various methods.

Create schema Movies

Create Nodes

Create node Movie
Properties: movie-ID key, title not null, release date, avg rating

Create node Person
Properties: ID key, name, date of birth, gender, type Not Null

Create node Genre
Properties: name key

Create node Rating
Properties: numeric rating key

Create node Monthly revenue
Properties: (year, month, movie-ID) key, income, avg rating

Create node RatingOfUserForMovie
Properties: (IDperson, IDmovie, numeric rating) key

Create Edges

Create edge ActedIn
Properties: roles
Start node Person (1,*n*)
End node Movie (1,*n*)

Create edge Directed
Start node Person (1,*n*)
End node Movie (1,*n*)

Create edge Produced
Start node Person (1,*n*)
End node Movie (1,*n*)

Create edge MonthlyRevenueForMovie
Properties
Start node Monthly revenue (0,*n*)
End node Movie (1,1)

Create edge MovieOfRatingFromUser
Start node RatingOfUserForMovie (0,*n*)
End node Movie (1,1)

Create edge UserOfRatingForMovie
Start node RatingOfUserForMovie (1,*n*)
End node Person (1,1)

Create edge RatingOfMovieAndUser
Start node RatingOfUserForMovie (1,*n*)
End node Rating (1,1)

Create edge GenreOf
Start node Genre (1,*n*)
End node Movie (1,*n*)

Figure 6. Data definition language (DDL) of the schema. This DDL can be added to a graph database system such as Neo4j.

References

1. S. Madden, "From Databases to Big Data," *IEEE Internet Computing*, vol. 16, no. 3, 2012, pp. 4–6.
2. H. Jing, E. Haihong, and L. Guan, "Survey on NoSQL Database," *Proc. 6th Int'l Conf. Pervasive Computing and Applications* (ICPCA), 2011; doi:10.1109/ICPCA.2011.6106531.
3. D. Hsieh, "NoSQL Data Modeling," Ebay tech. blog, 10 Oct. 2014; bit.ly/2hl1mI0.
4. A. Nayak, A. Poriya, and D. Poojary, "Type of NoSQL Databases and Its Comparison with Relational Databases," *Int'l J. Applied Information Systems*, vol. 5, no. 4, 2013, pp. 16–19.
5. A. Badia and D. Lemire, "A Call to Arms: Revisiting Database Design," *ACM SIGMOD Record*, vol. 40, no. 3, 2011, pp. 61–69.
6. P. Atzeni et al., "The Relational Model is Dead, SQL Is Dead, and I Don't Feel So Good Myself," *ACM SIGMOD Record*, vol. 42, no. 2, 2013, pp. 64–68.
7. K. Kaur and R. Rani, "Modeling and Querying Data in NoSQL Databases," *Proc. IEEE Int'l Conf. Big Data*, 2013; doi:10.1109/BigData.2013.6691765.
8. R. Angels, "A Comparison of Current Graph Database Models," *Proc. Data Engineering Workshops* (ICDEW), 2012, pp. 171–177.
9. I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*, O'Reilly Media, 2015.

***Noa Roy-Hubara** is an MSc student in 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 has recently submitted her thesis. Contact her at nro@post.bgu.ac.il.*

***Lior Rokach** is a professor in and the current chair of the Department of Software and Information Systems*

Engineering at Ben-Gurion University of the Negev, Israel. His main research interests are machine learning, recommender systems, cybersecurity, and information retrieval. Rokach received a PhD in industrial engineering from Tel Aviv University. Contact him at liorrk@post.bgu.ac.il.

***Bracha Shapira** is a professor of software and information systems engineering at Ben-Gurion University of the Negev (BGU), Israel. Her main research areas are information retrieval, recommender systems, cybersecurity, and data mining. Shapira received a PhD in information systems engineering from BGU. Contact her at bshapira@bgu.ac.il.*

***Peretz Shoval** is a professor emeritus of software and information systems engineering at Ben-Gurion University of the Negev, Israel, and head of the information systems program at Netanya College. His research interests include conceptual database modeling, information systems analysis and design methods, and information retrieval and filtering. Shoval received a PhD in information systems from the University of Pittsburgh. Contact him at shoval@bgu.ac.il.*

Take the CS Library wherever you go!

IEEE Computer Society magazines and Transactions are now available to subscribers in the portable ePub format.

Just download the articles from the IEEE Computer Society Digital Library, and you can read them on any device that supports ePub. For more information, including a list of compatible devices, visit

www.computer.org/epub

IEEE IEEE computer society