Dourhri *et al. Journal of Big Data* (2026) 13:114
https://doi.org/10.1186/s40537-026-01425-5

Journal of Big Data

RESEARCH Open Access

GRAPHDESIGN: a model-driven approach for a query-oriented graph database design

Ahmed Dourhri[1], Mohamed Hanine[1*], Hassan Ouahmane[1], Ernesto Bautista Thompson[2,3,4], Daniel Gavilanes[2,5,6] and Imran Ashraf[7*]

*Correspondence:
Mohamed Hanine
hanine.m@ucd.ac.ma
Imran Ashraf
ashrafimran@live.com
[1]LTI Laboratory, National School of Applied Sciences, Chouaib Doukkali University, El Jadida 24002, Morocco
[2]Universidad Europea del Atlántico, Isabel Torres 21, Santander 39011, Spain
[3]Universidad Internacional Iberoamericana, Campeche 24560, Mexico
[4]Universidad Internacional Iberoamericana Arecibo, Puerto Rico 00613, USA
[5]Universidade Internacional do Cuanza, Cuito, Bie, Angola
[6]Universidad de La Romana, La Romana, Dominican Republic
[7]Department of Information and Communication Engineering, Yeungnam University, Gyeongsan 38541, Republic of Korea

Abstract

The wide use of social networks is generating a huge quantity of data, structured and unstructured. With increased data volume and complexity, traditional databases became less efficient, which led to the development of NoSQL, a new generation of Database Management Systems. Despite the typical schema-optional nature of NoSQL databases, researchers have conducted studies on their design and emphasized the need for modeling techniques tailored to NoSQL data management. Graph-oriented databases, as a part of the NoSQL ecosystem, are gaining popularity because of their querying efficiency for complex connected data. Multiple aspects must be considered when modeling a graph-oriented database, generally a NoSQL database, including the relevant domain data and how these data will be accessed. Although several academic works have proposed modeling processes for graph-oriented databases, most of them do not rely on metamodeling to automate the design process and often leave important steps insufficiently specified. This study addresses the limitations observed in existing graph database design approaches by proposing a modeling guide that considers key requirements of graph-oriented database design. The proposed approach relies on Model-Driven Architecture to structure the design process and automate the generation of multiple models. By combining metamodeling, query-driven design, and formal specification, the approach improves model quality and maintainability while explicitly integrating performance considerations into the schema design process. We conducted an experiment on graph databases containing millions of nodes and relationships. Experimental results indicate that the proposed design achieves query execution time reductions ranging from 17% to 99% compared to the baseline schema, and remains more efficient than other representative state-of-the-art design approaches, while introducing an additional space overhead of approximately 62%. This balance highlights the method's practicality and its suitability for large graph databases.

Keywords NoSQL, Graph databases, Big data, MDA, Indexing, Database modeling, Database design, Query-driven design, Workload-aware design

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

Springer

Introduction

The expansion of user-generated content has led to a significant increase in the volume and variety of data being created, modified, analyzed, and stored. In addition, a huge quantity of data has been produced due to new sources, such as sensors, monitoring systems, and automated trackers. The enormous volumes of data, usually referred to as Big Data, present novel challenges and possibilities in terms of storage, processing, and archiving [1].

While relational databases remain generally effective for structured and stable data, they face limitations when dealing with highly connected data, evolving schemas, and large-scale workloads, where frequent joins and schema rigidity can negatively impact performance and adaptability. To address these limitations, NoSQL database systems have emerged, among which graph-oriented databases have gained significant attention. Graph databases are particularly well suited for managing highly connected data using traversal-based query, making them attractive for domains such as social networks, recommendation systems, fraud detection, and e-commerce platforms. [2].

Graph databases are grounded in graph theory and represent data as a graph structure composed of nodes and relationships. In this paradigm, nodes correspond to real-world entities, while relationships capture the semantic connections between them. Most modern graph database management systems adopt the Labeled Property Graph (LPG) model, in which both nodes and relationships are enriched with labels and properties, enabling the storage of domain-specific information. This model naturally supports traversal-based query execution and is particularly effective for applications involving strongly connected data. As a result, graph databases have been widely adopted in practice, with systems such as Neo4j, Memgraph, and GraphDB being among the most commonly used graph-oriented DBMS (Database Management system(s)) [3].

Despite their schema-flexible nature, design and modeling remain critical issues in graph databases and the NoSQL paradigm in general. Atzeni [4] argues that NoSQL database modeling can have beneficial effects and defines two lines of research. First, the diversity of models and systems could pose challenges for stakeholders in the database field. Data modeling approaches derive their legitimacy from the need to standardize access. Secondly, approaches may be defined at both the logical and physical levels using data models. Likewise, Kaur and Rani [5] claim that NoSQL database modeling is still important to understand how to store data correctly. For graph-oriented databases, modeling decisions, such as node granularity, relationship orientation, and attribute placement, directly influence traversal efficiency, query execution time, and maintainability.

Several studies have shown that poorly designed schemas can negate the performance advantages of graph databases, highlighting the need for well-defined design methodologies rather than ad hoc modeling practices. Shoval et al. [6] introduce GDBS, one of the earliest methods for designing graph databases from ER diagrams. The approach focuses on transforming data-related requirements (entities, relationships, cardinalities) into a logical graph schema. ER constructs are mapped to nodes and edges through explicit rules, producing a clear structural graph model. In a later work [7], Shoval et al. extend GDBS by explicitly integrating functional requirements expressed as queries into the design process. Data-related requirements are captured via UML class diagrams, while queries are used to refine and optimize the graph schema through transformation rules.

Existing design approaches largely rely on manual heuristics and guidelines, making the process difficult to reproduce. Model-Driven Architecture (MDA) addresses these limitations by automating schema design and reducing human intervention [8]. MDA structures system specification into three abstraction levels–Computation-Independent (CIM), Platform-Independent (PIM), and Platform-Specific (PSM)–and relies on metamodeling and transformation rules to systematically derive lower-level models from higher-level specifications [9].

In this context, Akoka et al. propose an MDA-based forward engineering approach for graph databases that explicitly incorporates the four Big Data dimensions [10]. Starting from an ER conceptual model, the method defines transformations toward logical and physical graph schemas, with a particular focus on volume-driven scalability evaluation.

Similarly, León et al. present an approach relying on model-to-model transformation from UML class diagrams to Labeled Property Graphs. They define mapping rules for classes, associations, and attributes, ensuring semantic preservation and portability across graph DBMSs [11].

Moreover, in the coming years, data-intensive applications will increasingly operate in a strongly connected world [12], where large volumes of interrelated data must be efficiently exchanged and analyzed, making graph-oriented DBMS a natural choice. While several works provide valuable insights into schema generation and integrity enforcement, they did not combine query-driven design with model-driven architecture and formal specification in the context of graph databases for performance optimization, nor did they leverage these principles to ensure conformity, improve design rigor, and support automated and maintainable schema evolution. This situation highlights the need for a structured and coherent design guide that brings together key modeling principles for graph databases.

In this context, this paper introduces GRAPHDESIGN, a model-driven and query-aware design methodology for graph-oriented databases. Rather than claiming exhaustiveness, GRAPHDESIGN aims to consolidate a set of relevant design concepts within a framework that can serve as a reference for both researchers and practitioners.

The methodology starts from data-related requirements expressed as a class diagram, which is adjusted to better fit the graph paradigm and transformed into a Graph Logical Data Model (GLDM) through mapping rules. Functional requirements, captured as data access queries, are then used to refine the GLDM and obtain a query-driven schema version. Finally, an experimental evaluation is conducted to assess the impact of the proposed design choices.

The rest of the article is organized as follows: Section "Related Works" reviews the related works. Then, Section "Proposed methodology" details the different steps of the proposed methodology. Section "Results and Discussion" assesses the proposal and its impact on query execution speed and memory space. Finally, Section "Conclusions" is dedicated to the conclusion and future work.

Related works

With the emergence of NoSQL systems, a significant number of research works have addressed the problem of data modeling and schema design beyond the relational paradigm. Due to the diversity of NoSQL families, including key-value, column-oriented, document-oriented, and graph databases, several approaches have been proposed

to guide schema design according to data characteristics, access patterns, and performance requirements. These works differ in their design objectives, abstraction levels, and degrees of automation, reflecting the absence of a unified design methodology for NoSQL databases. Table 1 summarizes significant studies developed for the design of the NoSQL databases, including the graph-oriented databases.

Some academic works have investigated schema design methodologies for non-graph NoSQL databases. Many proposals adopt workload-aware or model-driven strategies to guide schema design and improve execution efficiency. Chebotko et al. [13] bring a big data modeling methodology for column-oriented databases. The authors defined data modeling principles and mapping rules to guide logical and physical data modeling for Cassandra DBMS.

In the same direction, Mukherjee et al. [14] present an ontology-driven methodology for designing document-oriented databases, integrating domain semantics, sample queries, and performance constraints to optimize MongoDB schemas for large-scale health data. Similarly, Chen et al. [15] present a workload-driven approach for aggregate-oriented NoSQL databases, showing how query patterns can be exploited to generate tailored schemas that improve performance while balancing scalability and consistency requirements.

Model-driven techniques have also been widely explored to structure NoSQL schema design. Dourhri et al. [2] propose an MDA-based approach for key-value databases that relies on functional requirements to automatically derive physical schemas, while Mortadelo [16] provides a model-driven process supporting both column and document stores, explicitly considering query frequency during schema generation. In a broader perspective, Fernández Candel et al. [17] introduce a unified metamodel for relational and NoSQL databases, aiming to bridge logical and physical representations and facilitate multi-model database tooling.

Collectively, these works demonstrate that for non-graph NoSQL systems, various strategies can be employed to achieve effective schema design, as data structures must be carefully organized to meet performance requirements. However, these approaches are primarily tailored to data models in which access paths are predefined and therefore cannot be directly transposed to graph databases, where data relationships, traversals, and navigation patterns play a fundamental role. Consequently, the following paragraphs focus on schema design approaches for graph databases.

Early contributions include the work of De Virgilio et al. [18], who proposed a methodology based on an oriented ER diagram followed by a partitioning process. While this approach represents one of the first attempts to structure graph database design, it primarily focuses on domain data modeling and does not explicitly integrate access queries or workload considerations into the schema construction process.

Model-driven approaches have also been explored to support systematic transformations toward graph databases. Abdelhedi et al. propose an MDA-based approach that transforms UML class diagrams into NoSQL physical models through an intermediate generic logical model compatible with column-, document-, and graph-oriented systems [19]. From their part, Akoka et al. [10] propose another MDA-based framework that starts from an enriched ER model and generates Neo4j schemas through a sequence of conceptual and logical meta-models. Their approach partially considers big data characteristics, particularly data volume, during physical model generation. However, other

Table 1 NoSQL database design approaches

Study	Year	Family[1]	Conceptual	Logical	Physical	Query driven	Automatic transformation	Evaluation
[18]	2014	G	Chen ERD notation	Graph logical data model	Neo4j ArangoDB	No	Yes	Yes
[13]	2015	Col	Chen ERD notation	Chebotko diagram	Cassandra	Yes	Yes	No
[24]	2016	G	-UML class diagram -OCL constraints	-GraphDB model -Gremlin query language	Blueprint API Java	No	Yes	No
[10]	2017	G	ERD	Graph logical data model	Neo4j	No	Yes	No
[19]	2017	Doc G Col	UML class diagram	NoSQL generic logical model	Cassandra MongoDB Neo4j	No	Yes	Yes
[20]	2018	G	Chen ERD notation	Graph logical data model	—	No	Yes	No
[21]	2019	G	GRAPHD (custom model)	—	Neo4j	No	No	No
[16]	2021	Col Doc	ER	-Document metamodel -Column metamodel	MongoDB Cassandra	Yes	Yes	Yes
[17]	2022	All	UML class diagram	Unified logical metamodel	Neo4j MongoDB HBase Redis MySQL	No	Yes	Yes
[7]	2023	Doc G	UML class diagram	-Document data logical schema -Graph data logical schema	Neo4j	Yes	Yes	Yes
[25]	2023	KV	UML class diagram	Key-value logical data model	Redis	Yes	Yes	Yes
[22]	2023	G	Starts from logical level	PG-Schema (data and constraints)	Generic property graph DBMS	No	No	Yes
[11]	2024	G	UML class diagram	Labeled Property Graph Model	Neo4j	Yes	Yes	Yes
[23]	2025	G	Chen E/R notation extended	E/R Graph schema (subclass of PG-Schema)	Neo4j	No	No	Yes

[1]**Col**: Column; **Doc**: Document; **G**: Graph; **KV**: Key-Value; **All**: All NoSQL families

aspects such as velocity, variety, and veracity, as well as query-driven optimization, are not fully incorporated into the design process. Similarly, De Sousa et al. [20] emphasize integrity constraints in graph database modeling without explicitly integrating workload queries into the schema optimization process.

Other works focus on improving conceptual expressiveness and formalization of graph schemas. Cordeiro et al. [21] introduce GRAPHED, a visual conceptual modeling notation for graph databases that supports labeled nodes, edges, and cardinality constraints. While GRAPHED improves schema readability and conceptual clarity, it remains limited to the conceptual level and does not provide automatic transformations or query-driven design mechanisms. In a similar vein, Angles et al. [22] propose PG-Schema, a formal schema language for property graphs that introduces types and key constraints to enhance standardization and schema validation. Building upon PG-Schema, Skavantzos et al. [23] introduce Entity/Relationship Graphs, which adapt classical E/R modeling concepts to property graphs by enforcing redundancy-free designs through constraint semantics. However, both PG-Schema and E/R Graphs primarily focus on structural correctness and constraint definition rather than workload-aware or performance-oriented schema design.

Some approaches attempt to bridge conceptual modeling and functional requirements in the context of graph databases. León et al. propose UML4LPG, a model-driven process to transform UML class diagrams into Labeled Property Graph schemas [11]. Grounded in MDA principles, the approach defines a set of transformation patterns that map UML constructs, such as classes, associations, compositions, and generalizations, into graph elements while preserving semantic equivalence. A key contribution of this work is the explicit consideration of analytical requirements: queries are identified upfront and used to classify UML elements according to their analytical relevance, which influences the selection of transformation patterns.

Shoval et al. [7] propose a two-stage approach for designing graph and document-oriented databases starting from UML class diagrams. In the first stage, data-related requirements are captured using UML, after which the conceptual model is adjusted to simplify the mapping toward NoSQL structures. In the second stage, functional requirements are explicitly introduced through SQL-like queries, which are used to refine the schema with the objective of reducing query execution time. This work is notable for acknowledging the importance of workload queries in NoSQL schema design and for integrating them early in the modeling process. The approach does not define a clear set of graph-specific optimization rules, nor does it provide a systematic framework for translating recurring query patterns into structural design decisions.

To the best of our knowledge, existing approaches provide limited support for query-driven design in which workloads are expressed using graph-oriented query languages and systematically exploited to guide schema construction. Moreover, important design aspects, such as indexing strategies and structural constraints, are often treated in isolation or not explicitly integrated into a unified design process. In addition, most existing works do not rely on explicit metamodeling to ensure model conformity across design stages and to facilitate the automation and reuse of mapping rules.

To address these limitations, we propose GRAPHDESIGN, a query-driven and model-driven process for graph database design. The approach introduces a Cypher-like query specification to capture access patterns in a form suitable for graph stores, relies on

explicit metamodels and well-defined mapping rules within an MDA framework to support automation, and provides a formal specification that strengthens the rigor and consistency of the proposed methodology.

Proposed methodology

The first subsection presents the general elements of the proposed methodology for designing graph databases. The next subsections provide more detail on these elements. In addition, a dedicated subsection introduces a running example based on an e-commerce database, including its UML-based conceptual data model, which is used throughout the paper to illustrate the methodology.

General overview

Figure 1 describes GRAPHDESIGN, the proposal for graph database design. GRAPHDESIGN uses the MDA approach, which relies on metamodeling to define the components and transitions between the different models.

At the conceptual level, the proposed approach starts with a UML class diagram that captures the data-related requirements of the application, including classes, attributes, and relationships. This initial model, referred to as the original class diagram, is then systematically adjusted to comply with graph-specific constraints, resulting in an adjusted class diagram that is readily transformable into the graph paradigm. Based on this adjusted model, a GLDM is generated, independently of any particular graph DBMS, and serves as a unified representation of data-related requirements.

However, designing efficient graph databases requires more than data modeling alone. In GRAPHDESIGN, functional requirements are explicitly captured through a query model, which reflects traversal patterns and access behaviors specific to graph databases. The query model is used to refine the initial GLDM and produce a query-driven GLDM, optimized for graph-oriented query execution. The GLDM thus acts as an intermediate layer between the conceptual and physical levels, embedding graph-specific semantics while preserving portability across graph DBMS. Finally, the refined logical model is transformed into a physical schema targeting a selected graph database management system.

Overall, the GRAPHDESIGN methodology proceeds as follows:

- **Conceptual modeling**: data-related requirements are first captured using a UML class diagram representing the application domain.
- **Graph Logical modeling**: the class diagram is then adjusted to retain only graph-compatible constructs, namely classes and binary or reflexive associations, yielding

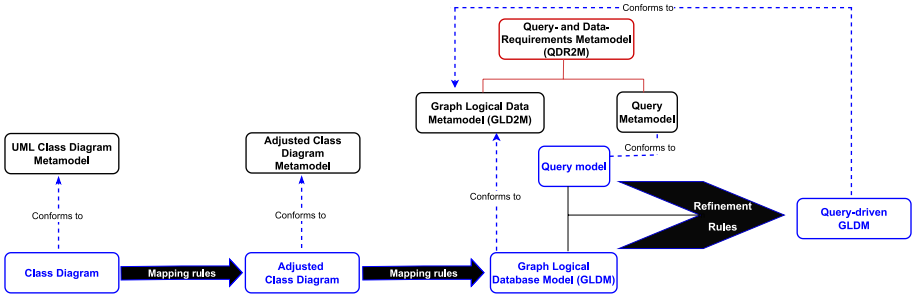


Fig. 1 Transformation process of GRAPHDESIGN approach

an adjusted class diagram. Afterwards, the adjusted class diagram is transformed, via a model-to-model transformation, into a Graph Logical Data Model that encodes graph-specific concepts independently of any DBMS.

- **Query-driven refinement**: functional requirements are captured through a query model describing CRUD and traversal queries; these queries are used to refine the GLDM and generate a query-driven GLDM, optimized for graph query execution.
- **Physical generation**: finally, a model-to-text transformation produces the physical schema targeting the selected graph database management system.

Running example

To illustrate the methodology, this work relies on an example inspired by an e-commerce platform. Table 2 represents its data dictionary while Figure 2 presents a conceptual data model of this platform, which is used to exemplify the different design principles.

As shown in the model, the platform manages *Product, Order, Client, Bank card, and Seller* as principal classes. The model was drawn according to the following rules:

- **Clients and orders**: A Client can create an Order that contains one or more Products, each supplied by a specific Seller. Another Client may pay for an Order that was not created by them. Each Client is described by a login, password, phone number, and account creation date. In addition, every Client must provide a residential address for order delivery.
- **Types of clients**: A Client may be either a Person or a Company. A Person Client is defined by their first and last name, while a Company Client requires a company name and government identification code.
- **Order context**: Each Order is associated with the device used to place it. The system stores the device's IP address, web browser, and operating system.
- **Products and sellers**: Multiple Sellers may provide the same Product. The system records the seller's available stock and warranty duration in months. Each Seller also sets the unit price of their Products and may apply a discount rate.
- **Product categories**: Every Product belongs to a Category, defined by its name and value-added tax (VAT) rate.

This example gathers the different relevant concepts for graph-oriented data modeling, namely data and different types of relationships as well as various query patterns. We choose this example to demonstrate key graph database design principles in a clear and practical context.

This study case contains some traditional data access, but it also contains other queries suitable for graph context (e.g. customer behavior patterns). Nevertheless, the different principles remain transposable to other use cases whenever data modeling for a graph database is required.

Class diagram

As part of the UML standard, the class diagrams model the real world as a collection of classes and relationships. These diagrams are frequently used by developers and researchers for data modeling.

Original class diagram

The first step is to create the class diagram, which should respect the specifications stated in the class diagram metamodel. Figure 3 is a metamodel for class diagrams, which are considered instances [26].

A class diagram is defined as a tuple: $CD = (C, A, R)$ where:

1. A is a finite set of attributes. Each attribute $a \in A$ is defined by the triple

$$a = (attribName, attribType, isUnique,$$

where *attribName* returns the attribute name, *attribType* denotes its data type, and $isUnique \in \{true, false\}$ specifies whether the attribute is unique.

2. C is a finite set of classes. Each class $c \in C$ is characterized by the following functions:

- a naming function: $cname : C \to String$;
- a function associating each class with the set of its attributes: $attrs : C \to \mathcal{P}(A)$.
- $Inc(c) = \{\, r \in R \mid c \in classes(r) \,\}$ denotes the set of relationships incident to class *c*;

3. R is a finite set of relationships connecting classes.

- Each relationship $r \in R$ connects a finite set of classes $classes(r) \subseteq C$.
- Each relationship is characterized by a naming function: $rname : r \to String$;
- The set of relationships R is composed from associations, aggregations, compositions, and inheritance relationships denoted respectively R_{assoc}, R_{aggr}, R_{comp}, and R_{inh}.

4. The set of associations R_{assoc} is itself decomposed into reflexive, binary, and n-ary associations denoted respectively $R^{\text{ref}}_{\text{assoc}}$, $R^{\text{bin}}_{\text{assoc}}$,and $R^{\text{nary}}_{\text{assoc}}$.

- Associations connect classes through ***association ends***.
  - An association end is defined as a triple $e = (c, m, r)$ where $c \in C$ is a participating class, $m \in Mult$ is a multiplicity constraint, and $r \in String \cup \{\perp\}$ is an optional role name, with $\perp$ denoting the absence of role information.
  - Multiplicity constraints are expressed as intervals *m..M*, where *m* and *M* respectively denote the minimum and maximum number of participations of a class in a relationship, with *M* possibly unbounded.
  - Each association $r \in R_{\text{assoc}}$ is characterized by its name *aname* and its optional set of attributes $attrs(r) \subseteq A$.

  - A **binary association** is denoted by $a_b = (aname, e_1, e_2, attrs, d)$ where: *aname* is the identifier name of the association in the model;
  - e_1 and e_2 are two association ends, each defined as a triple (*c*, *m*, *r*) specifying the participating class, its multiplicity, and its optional role;
  - and *attrs* is the set of attributes specific to the association, if it carries information ($\perp$ for an empty set).

Table 2 Data dictionary

Data name	Description	Type	Size
login	client user name	String	30
password	password	String	14
phone	phone number (national numbering)	Integer	12
creationDate	creation date of the client account	Datetime	—
firstname	first name of an individual client	String	20
lastname	his last name	String	20
name	company name of a business client	String	40
gov_id	company government identifier	Integer	—
addr_id	address identifier	String	12
road	road where the address is located	String	25
district	district	String	20
town	town	String	20
zipcode	zipcode	Integer	—
oid	order identifier	Integer	—
odatetime	date and time of an order	Datetime	—
shippingFee	shipping fees amount of an order	Float	—
total	total order amount (including shipping fees and taxes)	Float	—
sequenceNumber	sequence number for the order of a client	Integer	—
pm_number	payment number	Integer	—
pm_datetime	date and time of the payment	Datetime	—
idAccess	identifier number of an access operation	Integer	—
ip	public IP address of a device	String	15
webBrowser	web browser used to access the system	String	20
OSNameVersion	Operating system used in the remote access	String	20
timeAccess	date and time of a client access in the system	Datetime	—
card_number	bank card number (16 digits)	Integer	—
expireDate	date of expiration	Date	—
cvc	card verification value	Integer	—

Table 2 (continued)

Data name	Description	Type	Size
status	bank card status (disabled or enabled)	Boolean	—
bank	bank issuing the card	String	20
interestRate	credit card interest Rate	Float	—
limit	financing limit associated with the credit card	Float	—
balance	current account balance associated with the debit card	Float	—
label	debit card label (Visa,Mastercard,..)	String	16
reference	product reference	String	10
designation	designation of a product	String	30
description	its description	String	100
code	product category code	String	4
cname	product category name	String	20
vatRate	value added tax rate of the category	Float	—
orderedQty	quantity purchased of a product in an order from a seller	Integer	—
unitPrice	unit price applied by the seller for a product in an order	Float	—
discountRate	discount rate given by the seller for a product in an order	Float	—
seller_gov_id	government identifier of the seller company	Integer	—
company_name	name of the seller company	String	40
inStockQty	available quantity of a product in the seller's stock	Integer	—
warranty	month number of warranty	Integer	—

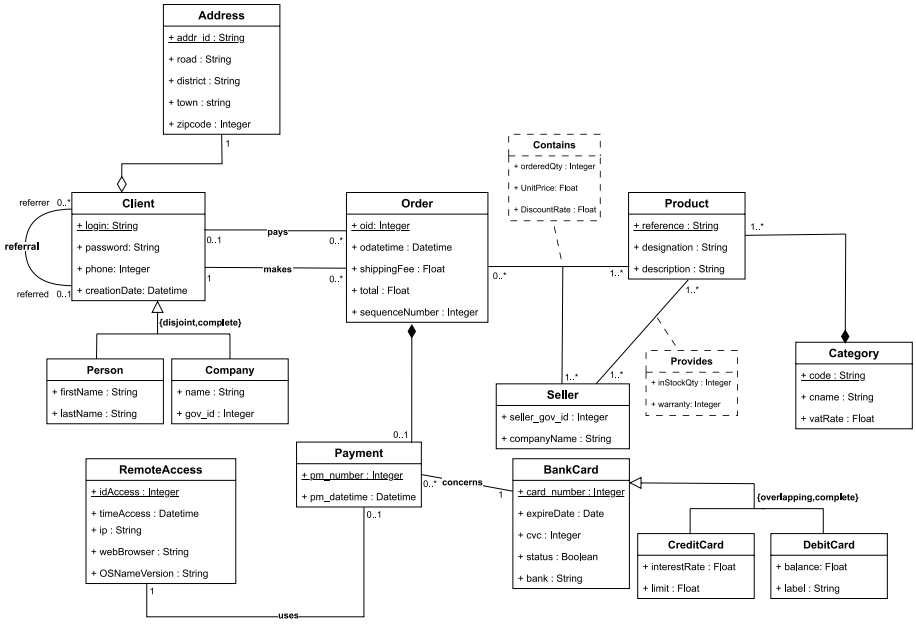


Fig. 2 UML class diagram for an e-commerce database

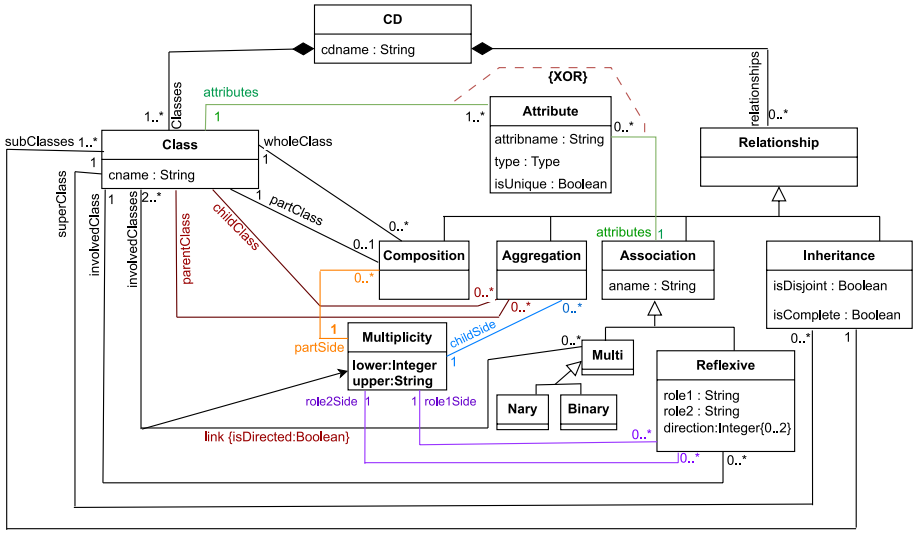


Fig. 3 Class diagram metamodel

– d is a direction indicator such that:
  * $d = 0$ denotes a bidirectional association (default case),
  * $d = 1$ denotes a directed association from extremity e_1 to extremity e_2,
  * $d = 2$ denotes a directed association from extremity e_2 to extremity e_1.

– A **n-ary association** is denoted by $a_n = (aname, e_{first}, e_{second}, E_{rest}, attrs)$ where *aname* is the name of the association,
– e_{first} and e_{second} are two distinct association ends serving as anchor points to identify the association,
– E_{rest} is a non-empty set of additional association ends, such that the total number of ends is three or more,

- *attrs* is the set of attributes carried by the association.

- A **reflexive association** is denoted by$a_r = (aname, c, (m_1, m_2), (r_1, r_2), attrs)$where *aname* is the name of the association,
- $c \in C$ is the class participating in the association with itself,
- m_1 and m_2 are the participation multiplicities associated with each of the two roles that an instance of class *c* may play in the relationship (e.g., *superior*/*subordinate*),
- r_1 and r_2 are the role names embodied in the relationship,
- and *attrs* is the set of attributes specific to the association.

5. An **inheritance** relationship is represented as a quadruplet:

$$inh = (super, Subs, isDisjoint, isComplete) \in R_{inh},$$

- $super \in C$ denotes the superclass, and $Subs \subseteq C$ is the set of its subclasses.
- The Boolean attributes *isDisjoint* and *isComplete* indicate whether the subclasses are mutually exclusive and whether the inheritance is complete.

6. A **Composition** relationship is defined as a quadruplet

$$r_{comp} = (whole, part, m_{whole}, m_{part}) \in R_{comp},$$

- $whole \in C$ denotes the composite (whole) class and $part \in C$ denotes the component (part) class.
- The multiplicity m_{part} specifies the number of instances of *part* that may be associated with a single instance of *whole*.
- The multiplicity m_{whole} is fixed to 1, expressing that each instance of *part* belongs to exactly one instance of *whole*.

7. An **Aggregation** relationship is defined as a quadruplet

$$r_{aggr} = (parent, child, m_{parent}, m_{child}) \in R_{aggr},$$

- $parent \in C$ denotes the aggregate (parent) class and $child \in C$ denotes the aggregated (child) class.
- The multiplicities m_{parent} and m_{child} specify, respectively, the number of instances of *parent* that may be associated with an instance of *child*, and the number of instances of *child* that may be associated with an instance of *parent*. by default, the multiplicity m_{parent} is assumed to be many ($*$), unless explicitly stated otherwise.

Adjusted class diagram

The initial class diagram may contain elements that are incompatible with a GLDM, which consists solely of object classes (represented by node labels) and binary relationships (represented by relationship types). Thus, we adjust the inheritance, aggregation, composition relationships, and n-ary associations using a series of adjusting rules AR1-AR3. Specifically, AR1 adjusts aggregation and composition relationships, AR2 handles inheritance relationships, and AR3 transforms *n*-ary associations into equivalent binary associations.

Figure 4 depicts the elements of the adjusted class diagram metamodel. An adjustment function $\mathrm{Adjust} : CD \rightarrow ACD$ is defined to maps a class diagram $CD = (C, A, R)$ to an adjusted class diagram $ACD = (C^*, A^*, R^*)$,where C^*, A^*, and R^* denote respectively the sets of classes, attributes, and relationships obtained after adjustment.

The function Adjust is the composition of three adjustment functions, AR1, AR2, and AR3, formally expressed as $\mathrm{Adjust}(CD) = \mathrm{AR3}(\mathrm{AR2}(\mathrm{AR1}(CD)))$. These adjustment rules are detailed in subsequent paragraphs.

Graph logical data metamodel

Figure 5 illustrates the graph logical metamodel, which is inspired by the Labeled Property Graph model. This metamodel is created to define the components of a graph-oriented database. We will employ the metamodel to ensure the computed logical model matches the graph context regardless of the used DBMS.

A graph logical data model is defined as a triple $G = (NL, P, RT)$, where:

- NL is a finite set of *node labels*, each label representing a node type in the graph and corresponding to a class in the adjusted class diagram.
- P is a finite set of *properties*, associated with node labels or relationship types, and representing the attributes of the original model.
- RT is a finite set of *relationship types*, connecting node labels, and representing associations of the adjusted class diagram.

Transition from conceptual level to logical level

It is performed using a set of rules, mainly based on the works of Shoval et al. [7] and De la Vega et al. [16]. These authors show how to use model-to-model transformations to convert a class diagram that represents a conceptual data model into a NoSQL logical model. The key idea behind these rules is to create a node label for each class as well

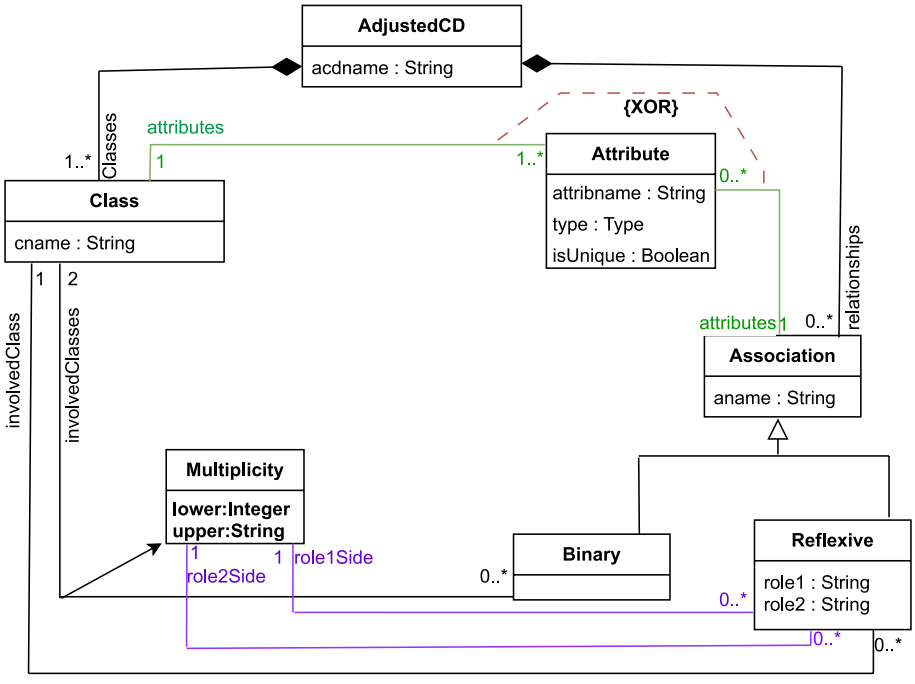


Fig. 4 Adjusted class diagram metamodel

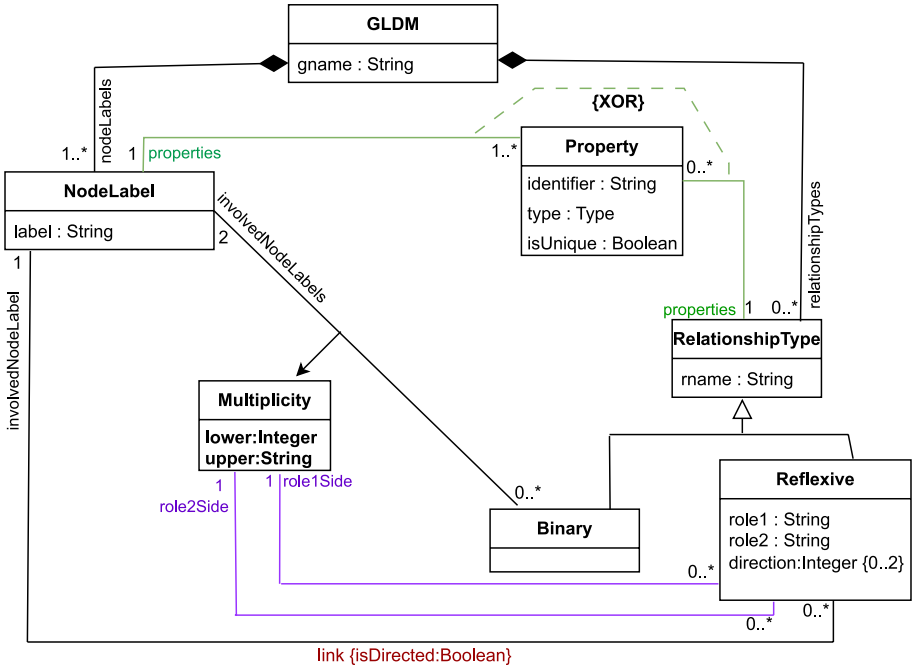


Fig. 5 Graph logical data metamodel

as a relationship type to support a binary or reflexive association. This idea respects the standards and best practices widespread in graph-oriented DBMS. The following sub-subsection describes the implementation strategy for this idea.

Class diagram adjusting

We adjust the class diagram to include only classes and non-n-ary associations and transform the other elements. The goal is to simplify the transition from the conceptual level to the logical graph model.

AR1: composition and aggregation

This rule rewrites aggregation and composition relationships into binary associations.

- **Composition.** For each composition $r_{comp} = (whole, part, m_{part}) \in R_{comp}$, linking a whole class $C1$ to a part class $C2$ with multiplicity m_{part} on the $C2$ side, AR1 introduces a new binary association

$$r_{new} = (\text{“}is_composed_of'', (C1, 1), (C2, m_{part}), \perp, 1)$$

The set of relationships is then updated as follows:

$$R^* \leftarrow (R^* \setminus \{r_{comp}\}) \cup \{r_{new}\}.$$

- **Aggregation.** For each aggregation $r_{aggr} = (C1, C2, m_{parent}, m_{child}) \in R_{aggr}$, linking a parent class $C1$ to a child class $C2$, AR1 introduces a new binary association $r_{new} = (\text{“}has'', (C1, m_{parent}), (C2, m_{child}), \perp, 1)$, where m_{parent} denotes the $C1$ multiplicity and m_{child} denotes the $C2$ multiplicity. The set of relationships is then updated as follows:

$$R^* \leftarrow (R^* \setminus \{r_{aggr}\}) \cup \{r_{new}\}.$$

When several aggregation relationships exist, the corresponding binary associations are distinguished by indexing their names (e.g., has_1, has_2,...) in order to avoid naming conflicts. The same convention is applied to composition relationships.

In both cases, the *whole–part* and *parent–child* semantics are preserved, while being represented as binary associations compatible with the graph paradigm. Figure 6 illustrates these two transformation rules.

AR2: inheritance

Two strategies are offered to transform each inheritance relationship

$$inh = (super, Subs, isDisjoint, isComplete) \in R_{inh},$$

where $super \in C$ denotes superclass and $Subs \subseteq C$ is a non-empty set of its subclasses.

- **Strategy A**: is applied if the disjointness and completeness constraints are satisfied in the relationship. If $isDisjoint = true$ and $isComplete = true$, the superclass *super* is preserved and the subclasses are removed, after performing the following operations:

1. *Attribute merging into the superclass.* All attributes of the subclasses are added to the superclass:

$$attrs(super) \leftarrow attrs(super) \cup \bigcup_{c \in Subs} attrs(c).$$

2. *Addition of a discriminating attribute type.* A new attribute a_{type} of type *string* is created and added to the superclass:

$$A^* \leftarrow A^* \cup \{a_{type}\}, \quad attrs(super) \leftarrow attrs(super) \cup \{a_{type}\}.$$

This attribute records, for each instance of *super*, the name of its original subclass.

3. *Redirection of subclass associations to the superclass.* For each subclass $c \in Subs$ and for each relationship $r \in Inc(c)$, a new relationship r' is constructed by replacing every occurrence of *c* with *super*, while preserving the remaining elements (name, multiplicities, and attributes):

$$R^* \leftarrow (R^* \setminus \bigcup_{c \in Subs} Inc(c)) \cup \{r' \mid \exists c \in Subs, \exists r \in Inc(c), r' = redirect(r, c, super)\}.$$

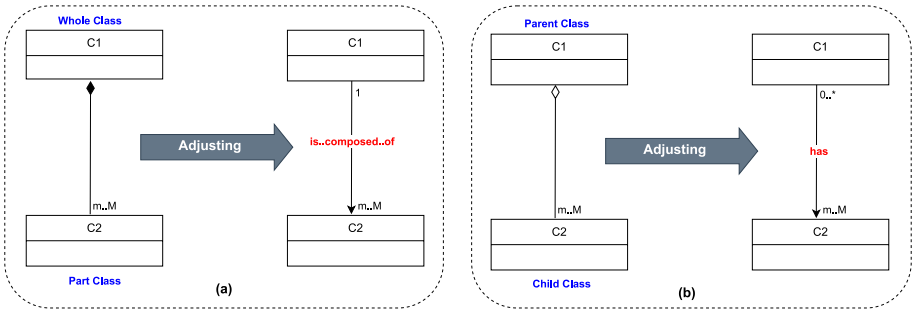


Fig. 6 Adjustment rules for composition and aggregation

The redirection function is defined as: $redirect : (r, c, s) \in R \times C \times C \rightarrow r' \in R$. This function performs a structural rewriting of a relationship by replacing every occurrence of class *c* with class *s*. The resulting relationship r' preserves the relationship name, roles, multiplicities, and associated attributes.

4. *Removal of subclasses.* Finally, the subclasses are removed from the set of classes:

$C^* \leftarrow C^* \setminus Subs.$

- **Strategy B**: is applied otherwise. In this case, the superclass is removed and the subclasses are preserved, after performing the following operations:

1. *Duplication of superclass attributes into the subclasses.* For each subclass $c \in Subs$, all attributes of the superclass are added to *c*:

$\forall c \in Subs, \quad attrs(c) \leftarrow attrs(c) \cup attrs(super).$

2. *Duplication of superclass associations into the subclasses.* For each relationship $r \in Inc(super)$ and for each subclass $c \in Subs$, a new relationship r_c is created by replacing every occurrence of *super* with *c*, while preserving the remaining elements:

$R^* \leftarrow (R^* \setminus Inc(super)) \cup \{r_c \mid \exists c \in Subs, \exists r \in Inc(super), r_c = redirect(r, super, c)\}.$

3. *Removal of the superclass.* Finally, the superclass is removed from the set of classes:

$C^* \leftarrow C^* \setminus \{super\}.$

The different strategies are illustrated by Figure 7.

AR3: n-ary associations

Each n-ary association $a_n = (aname, e_{first}, e_{second}, E_{rest}, A_n) \in R^{\text{nary}}_{\text{assoc}}$ involving $n \geq 3$ participating classes, is adjusted by applying the following series of operations:

1. *Creation of the pivot class.* A new pivot class C_A is created such that:

$C^* \leftarrow C^* \cup \{C_A\}, \quad \text{with } cname(C_A) = aname.$

2. *Transfer of association attributes to the pivot class.* All attributes of the *n*-ary association are transferred to the pivot class:

$A^* \leftarrow A^* \cup A_n, \quad attrs(C_A) \leftarrow A_n.$

3. *Creation of binary associations.* For each participating class C_i in the *n*-ary association a_n, AR3 introduces a binary association a_i linking the pivot class C_A to C_i with multiplicities m_A and m_i, respectively. m_i denote the multiplicity of C_i in the *n*-ary association, and m_A be a multiplicity defined by the model designer. Each binary association a_i can be named *aname* & $cname(c_i)$ where the character & is the concatenation operator.
4. *Removal of the n-ary association.* Finally, the *n*-ary association is removed from the set of relationships:

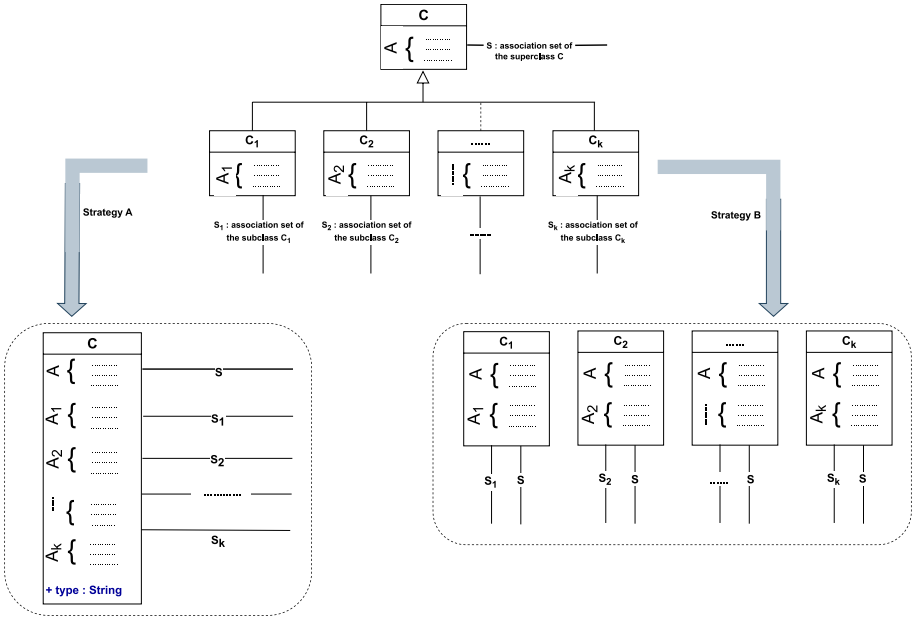


Fig. 7 Transformation of inheritance relationships

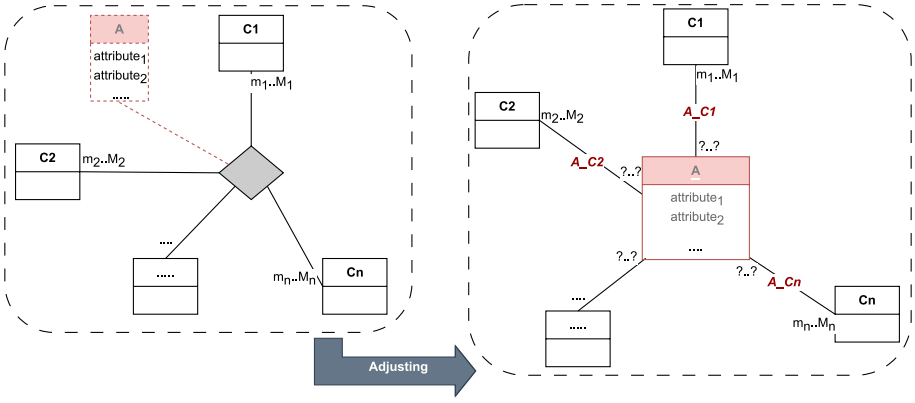


Fig. 8 Transformation of N-ary associations

$$R^* \leftarrow R^* \setminus \{a_n\}.$$

The different rules are illustrated by Figure 8.

Running example application

Taking the e-commerce class diagram as an example, we apply the different rules:

- The original class diagram contains two aggregations and composition relationships that must be transformed into three binary associations. The associations resulting from this transformation are colored red (for aggregation) and purple (for composition) in the adjusted version of class diagram;
- Furthermore, the class diagram contains two inheritance relationships, and the rule requires their transformation using an appropriate strategy. Note that the inheritance-adjusting results are colored green (if strategy A is used) or orange (else);
- Finally, the ternary association "Contains" is transformed into a class with the same name in the class diagram, carrying exactly the attributes defined on the original

association. This transformation is marked blue on the adjusted version of the diagram.

Figure 9 shows the adjustment of the original e-commerce class diagram.

Adjusted-CD to GLDM transformation

This transformation generates a graph logical data model from an adjusted class diagram *ACD*. The process produces a set of node labels and a set of relationship types according to the following principles:

- The GLDM model has the same name as the adjusted class diagram.
- A node label *nl* is created for each class *c* in the *ACD*. The attributes of *c* are transformed into properties of *nl*.
- Each association *assoc* in the *ACD* is transformed into a relationship type *rt* in the GLDM. If the association *assoc* carries attributes, these attributes become the properties of *rt*.

Formally, we define a transformation function $\mathrm{AcdToGldm} : ACD \rightarrow GLDM$ that produces a graph logical data model $GLDM = (NL, RT, P)$ from an adjusted class diagram $ACD = (C^*, A^*, R^*)$. This transformation is done through the following steps.

1. **Construction of the node label set** NL**.** Each class $c \in C^*$ is transformed into a node label $nl_c \in NL$:

$$NL = \{nl_c \mid c \in C^*\}, \quad label(nl_c) = cname(c), \quad properties(nl_c) = attrs(c).$$

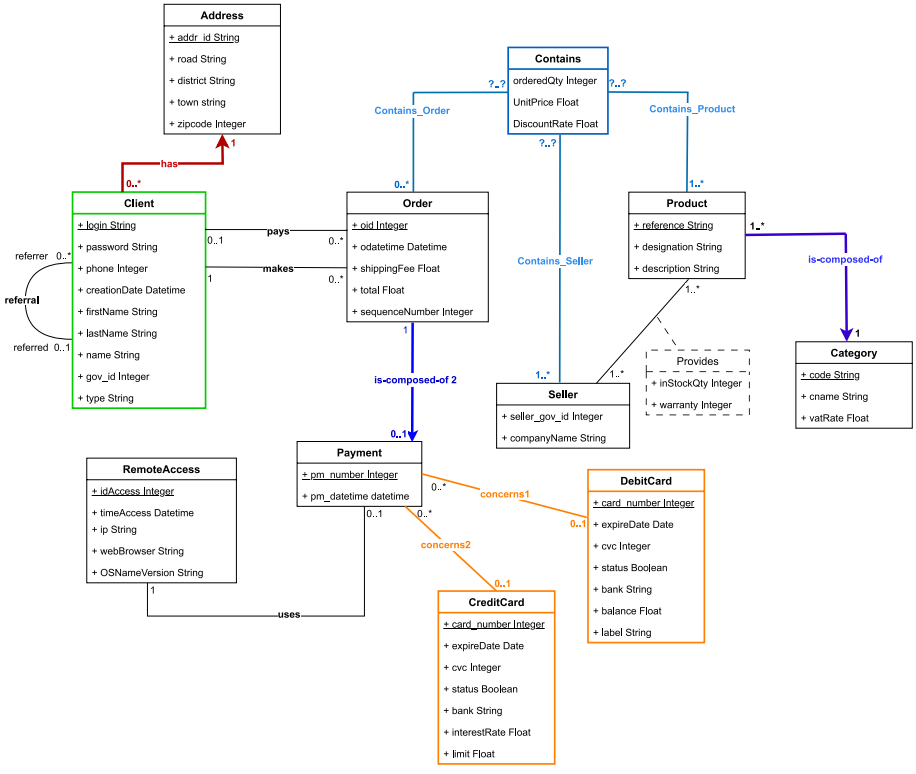


Fig. 9 Adjustment of the original e-commerce class diagram

2. **Construction of the relationship type set** RT**.** The set of relationship types, including binary and reflexive ones, is defined as $RT = RT_{bin} \cup RT_{ref}$ where:

   - For each binary association $a_b = (aname, e_1, e_2, attrs, dir) \in R^{*\ \text{bin}}_{\text{assoc}}$, a binary relationship type is defined as $rt = mapBin(a_b) \in RT_{bin} mapBin(aname, e_1, e_2, attrs, dir) =; (aname, e'_1, e'_2, properties(rt), dir)$
   - $properties(rt) = \{p_a \mid a \in attrs\} \subseteq P$;
   - each extremity $e'_i = (nl_{c_i}, m_i, r_i)$ is obtained by replacing class c_i with its corresponding node label nl_{c_i}.

   - For each reflexive association $a_r = (aname, c, (m_1, m_2), (r_1, r_2), attrs) \in R^{*\ \text{bin}}_{\text{assoc}}$, a reflexive relationship type is defined as $rt = mapRef(a_r) \in RT_{ref}$ $mapRef(aname, c, (m_1, m_2), (r_1, r_2), attrs) = (aname, nl_c, (m_1, m_2), (r_1, r_2), properties(rt))$
   - $properties(rt) = \{p_a \mid a \in attrs\} \subseteq P$
   - nl_c denotes the node label associated with class *c*.

3. **Construction of the property set** P**.** Each attribute $a \in A^*$ is transformed into a property. The set of properties of the GLDM is defined as $P = \bigcup_{nl \in NL} properties(nl) \cup \bigcup_{rt \in RT} properties(rt)$.

Thus, all classes are transformed into node labels, all associations are transformed into relationship types, all attributes become properties, and the multiplicities are preserved. Figure 10 presents the GLDM associated with the running example.

Data- and query- related metamodel

Given the importance of queries in NoSQL database design, including graph-oriented ones, a query model enhances the first version of GLDM. A query model, which contains a list of access queries, aims to represent different functional requirements of an application. We use the following queries for the running example to demonstrate how the methodology works:

Q1: Orders in which a client is involved in any role (making or payment).

Q2: Products of a seller, for which the warranty exceeds six months.

Q3: Company-type clients are making orders on a given date.

Q4: From a given order of a client, retrieve the next order.

Q5: Categories were purchased by a specific client.

Q6: List of disabled cards with the label "Master".

Q7: List of "Master" cards used on a given date, sorted by descending chronological order of payment time.

Q8: List of "Visa" cards from a given bank.

Q9: Clients of a given town.

Q10: The orders of a specific client, for which the shipping fees ranged from 100 to 10,000 currency units, were placed on a specific date.

Q11: Pairs of clients sharing the same credit card.

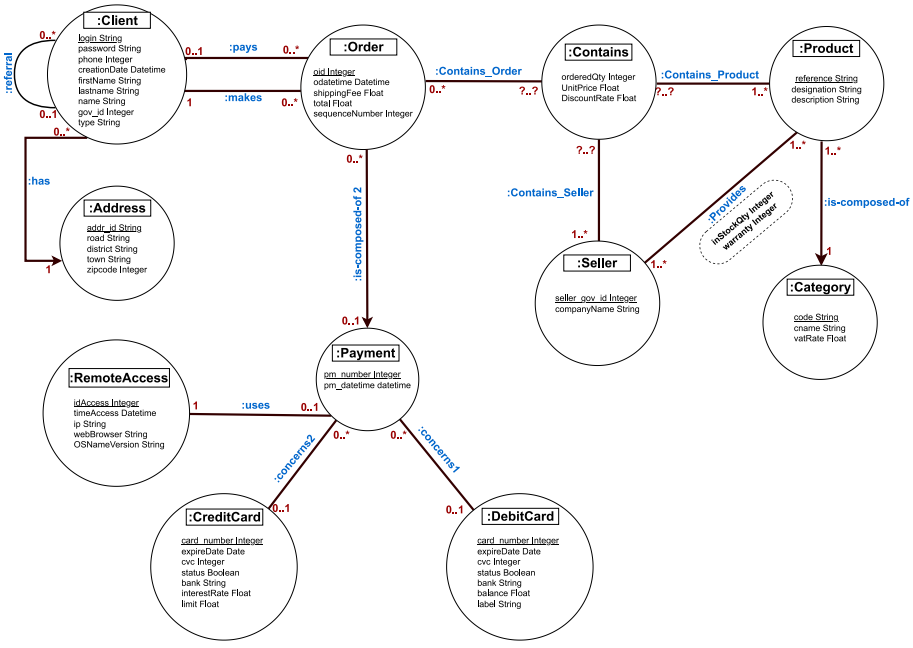


Fig. 10 Data-driven GLDM model

Q12: List of clients having paid orders from the same IP. The result must be sorted by IP address order.

Q13: The orders paid with cards owned by other clients.

Q14: Modify the VAT rate for a given category.

Q15: Change the status of a given banking card.

Q16: Change the warranty duration offered by a seller for a product.

Q17: Update a client's phone number.

Q18: Delete a product from the catalog.

Q19: Remove orders placed prior to 2014.

Figure 11 depicts the Data- and Query- Related Metamodel (denoted DQR2M), which includes the graph logical data metamodel in addition to the query metamodel that defines the components of a graph query. Each query is written on a GLDM whose structure is known previously. Moreover, a query can modify the GLDM structure. To avoid the complexity of working on separate models, we place the data and query-related requirements together. Accordingly, a DQR2M instance is a model that contains a GLDM model and a list of access queries.

The DQR2M supports various CRUD query types, but this paper only illustrates how to modify schemas based on read queries. We made this choice to avoid making the article too lengthy, especially considering that read queries are the most commonly used and time-consuming in NoSQL database design [27].

Formally, the DQR2M metamodel is defined as the pair:

$$DQR2M = (GLD2M, Q2M),$$

where *GLD2M* denotes the metamodel of the graph logical data model, and *Q2M* denotes the query metamodel defining the structure of access queries.

A DQR2M model instance is defined as:

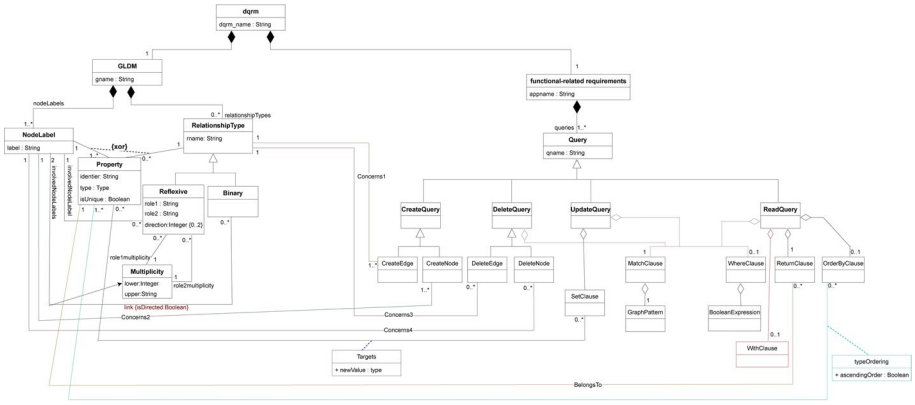


Fig. 11 Data- and query-related metamodel

$$DQRM = (G, Q),$$

where:

- $G = (NL, RT, P)$ is a graph logical data model conforming to *GLD2M*;
- $Q = \{q_1, q_2, \ldots, q_m\}$ is a finite set of access queries expressed over G and conforming to the query metamodel *Q2M*.

The integration of query requirements into the data model leads to an optimized graph model, denoted G_Q, defined as:

$$G_Q = (NL_Q, RT_Q, P_Q),$$

where NL_Q, RT_Q, and P_Q respectively denote the sets of node labels, relationship types, and properties resulting from the integration of the query workload Q into the initial graph model *G*.

In DQR2M, an access query is formally defined as a tuple:

$$q = (M, WR, WT, R, ORD),$$

where:

- $M \in GP$ is the `MATCH` clause, specifying a graph pattern to be matched over the graph model;
- $WR \in BoolExpr \cup \{\perp\}$ represents the optional `WHERE` clause, used to filter matched data through boolean conditions;
- $WT \in WithExpr \cup \{\perp\}$ represents the optional `WITH` clause, enabling intermediate result chaining;
- $R \subseteq P$ denotes the set of properties returned by the query;
- $ORD \in (P \times \{true, false\})^+ \cup \{\perp\}$ represents the optional `ORDER BY` clause, where each pair (*p*, *b*) specifies a sorting property *p* and an ordering direction (*true* for ascending, *false* for descending).

Model specification language

We developed a textual syntax to create models that align with the specifications of the different metamodels. This subsection describes how to represent the "ecommerce" running example using specification language at the conceptual and logical levels.

Conceptual class diagram

Figure 12 explains how to express the different class diagram elements at a conceptual level, while Figure 13 illustrates this syntax using the e-commerce running example.

- Classes are declared using the `class` keyword, and their attributes are defined between braces, with optional uniqueness constraints.
- Inheritance relationships are explicitly specified using the `inheritance` construct, together with the semantic properties `isDisjoint` and `isComplete`.
- Associations are declared using the `association` keyword, with explicit multiplicities and optional role names; reflexive and *n*-ary associations are handled through dedicated syntactic forms.
- Aggregation and composition relationships are also explicitly declared, with their corresponding parent–child or whole–part semantics and multiplicity constraints. For composition, the multiplicity on the side of part class is the one required. The multiplicity on the side of the whole class is implicitly *one*.

Graph logical data model

The rules presented in Figure 14 specifies the GLDM, which contains node labels and relationship types. Figure 15 shows how the node labels Client, Address, Order, and the Referral, pays, and Provides relationship types are expressed in the syntax.

Although multiplicity constraints are not natively enforced by most graph database management systems, they can play a role at the conceptual and logical design levels.

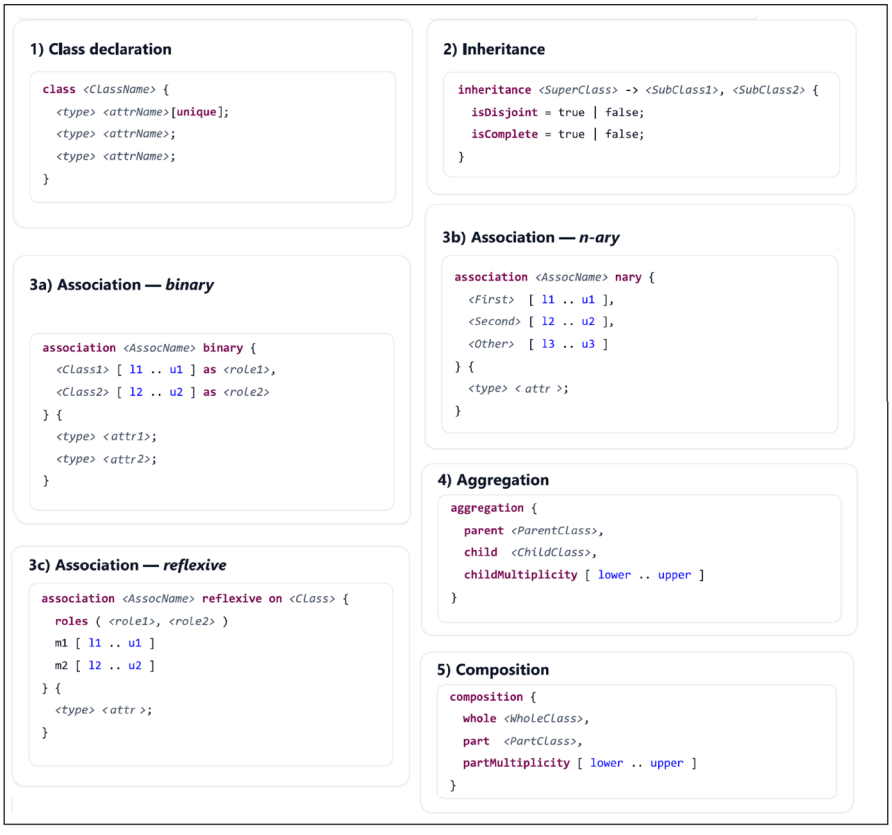


Fig. 12 Specification syntax for Class Diagram elements

```
class Client {
  string login unique;
  string password;
  int phone;
  datetime creationDate;
}
inheritance Client -> Person, Company {
  isDisjoint = true;
  isComplete = true;
}
aggregation {
  parent Address,
  child Client,
  childMultiplicity [1]
}
composition {
  whole Order,
  part Payment,
  partMultiplicity [0..1]
}
```

```
association PAYS binary {
  Client [0..1],
  Order  [0..*]
}
association CONTAINS nary {
  Order   [0..*],
  Product [1..*],
  Seller  [1..*]
} {
  int orderedQty;
  float unitPrice;
  float discountRate;
}
association REFERRAL reflexive on Client {
  roles (referrer, referred)
  m1 [0..*]  // referrer multiplicity
  m2 [0..1]  // referred multiplicity
}
```

Fig. 13 Example of a class diagram specification

Consequently, multiplicities are handled at the application layer and can provide valuable insights to future graph engines or design tools to enhance execution planning and structural optimization.

Graph query specification

Queries are expressed using the CYPHER-based clauses [28]. Figure 16 shows how to write, at the logical level, the first five queries in the listing using the dedicated syntax. The rest of the query model is defined and is consultable in an external repository [29].

The QUERY keyword specifies a query, followed by a name that identifies its purpose. Next, we use the following clauses to provide the different elements of a query:

- **MATCH**. It specifies the patterns we will search for in the database. A pattern expresses a traversal path composed of nodes and/or relationships, and it is used to explore the graph in order to search for the data we are interested in.
- **WITH**. Sometimes it is not practical or possible to do everything in a single match. We can chain together several matches using the WITH clause, piping the results of the previous query part into the next.
- **RETURN**.The query retrieves a set of nodes, relationships, or properties from it. These data can come from the objects that are in *MATCH* or *WITH* clauses;
- **WHERE**. This clause allows a query to filter data that meets a condition;
- **ORDERBY**. It specifies the properties involved in sorting a query result. The sorting can be performed either in ascending or descending order. Sometimes the order can be based on multiple criteria (a.k.a., multi-criteria ordering).

In the following, the? character will be used to represent unknown information in a query, such node labels, relationship types, properties, or variables, or a parameter value for a parameterized query.

a) Node label

```
nodeLabel <NodeLabelName> {
  <type> <propertyName> unique;
  <type> <propertyName>;
}
```

b) Relationship type — *binary*

```
relType <RelName> binary {
  <NodeLabel1> [ l1 .. u1 ] as <role1>,
  <NodeLabel2> [ l2 .. u2 ] as <role2>
} {
  <type> <relProperty>;
}
```

c) Relationship type — *reflexive*

```
relType <RelName> reflexive on <NodeLabel> {
  roles ( <role1>, <role2> )
  m1 [ l1 .. u1 ]
  m2 [ l2 .. u2 ]
} {
  <type> <relProperty>;
}
```

Fig. 14 Syntax of GLDM model elements

Query-based structural optimizations of the GLDM

This subsection defines a list of rules based on the works of De la Vega et al. [16] and Shoval et al. [7] for optimizing the structure of a GLDM using a query model. These rules respect best practices and standards used by professional community.

Let a logical data model $G = (NL, RT, P)$ and a query model $Q = \{q_1, q_2, \ldots, q_m\}$ defined over G.

1. The following notations are used in the definition of the different query-based structural optimization rules.

   - The cardinality of a set S is denoted by $|S|$.
   - $values(p)$ returns the set of possible values for a property p, or ∞ if this set is infinite;
   - $name(p)$ returns the identifier name of property p;
   - $e.p$ returns the property p of an element e, where $e \in \{NL, RT\}$.

```
nodeLabel Client {
  string login unique;
  string password;
  int    phone;
  datetime creationDate;
  string type;
}

relType REFERRAL reflexive on Client {
  roles ( referrer, referred )
  m1 [ 0..* ]
  m2 [ 0..1 ]
}

relType PAYS binary {
  Client [ 0..1 ],
  Order  [ 0..* ]
}
```

Fig. 15 Example of GLDM specification syntax

```
Query Q1_OrdersOfClient as
MATCH (c:Client)-[]-(o:Order)
WHERE c.login = ?
RETURN o
;

Query Q2_ProductsOfSellerWithWarranty as
MATCH (p:Product)-[pvd:Provides]-(s:Seller)
WHERE s.seller_gov_id = ? AND pvd.warranty > 6
RETURN p
;

Query Q3_CompanyClientsOrderedInDate as
MATCH (c:Client)-[:makes]-(o:Order)
WHERE c.type = "company" AND o.odatetime = ?
RETURN c
;

Query Q4_nextOrderOfClient as
MATCH (c1:Client)-[:makes]-(o1:Order), (c2:Client)-[:makes]-(o2:Order)
WHERE c1.login = ? AND c2.login = c1.login AND o2.sequenceNumber = o1.sequenceNumber + 1
RETURN o2
ORDER BY o2.sequenceNumber ASC
;

Query Q5_categoriesPurchasedByClient as
MATCH (c:Client)-[m:makes]-(o:Order)-[co:containsOrder]-(cnt:Contains)-
      [cp:containsProduct]-(p:Product)-[comp:is_composed_of]-(ct:Category)
WHERE c.login = ?
RETURN ct.code, ct.cname
;
```

Fig. 16 Example of query model (Queries Q1 to Q5)

- *properties*(*e*) returns the set of properties associated with an element *e*, where $e \in \{NL, RT\}$;
- *label*(*nl*) returns the label of a node label *nl*;
- *id*(*nl*) returns the identifier o *nl*;
- *rname*(*r*) returns the type name of a relationship type *r*;
- For a relationship type $rt \in RT$, *m* and *M* respectively denote its minimum and maximum multiplicities.
- Let two node labels $A, B \in NL$. *conn*(*A*, *B*) returns the set of relationship types connecting *A* and *B*, independently of their possible orientation, and is defined as: $conn(A, B) = \{\, rt \in RT \mid \exists e_1 = (A, m_1, r_1),\ e_2 = (B, m_2, r_2)$ such that $(rt = (rname, e_1, e_2, properties(rt))) \ \vee\ (rt = (rname, e_2, e_1, properties(rt)))\,\}$.
- For a query $q \in Q$, we define the following auxiliary functions: ∘ *wrProps*(*q*) returns the set of properties involved in the `WHERE` clause of *q*; ∘ *ordProps*(*q*) returns the set of properties involved in the `ORDER BY` clause of *q*; ∘ *M*(*q*) returns the graph pattern specified in the `MATCH` clause of *q*;

2. **Operations on Multiplicities** We consider multiplicity bounds $a, b \in \mathbb{N} \cup \{?, *\}$, where $*$ denotes an unbounded multiplicity and? denotes an unknown multiplicity.

   • Binary operations on two bounds *a* and *b*.

   – **Extended sum.** The extended sum of two bounds *a* and *b* is defined as:

$$a \oplus b = \begin{cases} * & \text{if } a = * \text{ or } b = *, \\ ? & \text{if } a =? \text{ or } b =?, \\ a + b & \text{if } a, b \in \mathbb{N}. \end{cases}$$

   – **Extended product.** It is defined for two bounds *a* and *b* as:

$$a \otimes b = \begin{cases} 0 & \text{if } a = 0 \text{ or } b = 0, \\ * & \text{if } a = * \text{ or } b = *, \\ ? & \text{if } a =? \text{ or } b =?, \\ a \times b & \text{if } a, b \in \mathbb{N}. \end{cases}$$

   – **Extended maximum.** The extended maximum of two bounds *a* and *b* is defined as:

$$emax(a, b) = \begin{cases} * & \text{if } a = * \text{ or } b = *, \\ ? & \text{if } a =? \text{ or } b =?, \\ \max(a, b) & \text{if } a, b \in \mathbb{N}. \end{cases}$$

   • *k*-ary operations over a family $(x_1, \ldots, x_k)$. Let $x_1, \ldots, x_k \in \mathbb{N} \cup \{?, *\}$ be *k* multiplicity bounds.

   – **Extended sum.** The extended sum over *k* bounds is defined as:

$$\bigoplus_{i=1}^{k} x_i = \begin{cases} * & \text{if } \exists i : x_i = *, \\ ? & \text{if } \exists i : x_i =?, \\ \sum_{i=1}^{k} x_i & \text{otherwise (all bounds are numeric).} \end{cases}$$

   – **Extended product.** The extended product over *k* bounds is defined as:

$$\bigotimes_{i=1}^{k} x_i = \begin{cases} 0 & \text{if } \exists i : x_i = 0, \\ * & \text{if } \exists i : x_i = *, \\ ? & \text{if } \exists i : x_i = ?, \\ \prod_{i=1}^{k} x_i & \text{otherwise (all bounds are numeric).} \end{cases}$$

– **Extended maximum.** The extended maximum over *k* bounds is defined as:

$$emax_{i=1}^{k} x_i = \begin{cases} * & \text{if } \exists i : x_i = *, \\ ? & \text{if } \exists i : x_i = ?, \\ \max\{x_1, \ldots, x_k\} & \text{otherwise (all bounds are numeric).} \end{cases}$$

Adding generic relationship type

Rule1. In cases where a query involves any type of relationship between two node labels NL_1 and NL_2, a generic relationship type between NL_1 and NL_2 should be added to the Graph Logical Data Model (in addition to the existing relationship types). In such a scenario, incorporating a generic relationship type between NL_1 and NL_2 would enhance the efficiency of this query. This relationship type can be named "participates". Figure 17 demonstrates, using a schema, how to apply this rule to a GLDM.

Let *Q* be a query model defined over a graph logical model $G = (NL, RT, P)$.

The rule is formally defined as follows:

$$\forall A, B \in NL \times NL,$$
$$\big(|conn(A,B)| > 1 \ \wedge \ (\exists q \in Q : isGeneric(A,B,q))\big) \ \Rightarrow \ RT := RT \cup \{rt_{new}\},$$

where $rt_{new} = (\text{“}participates'', (A,(m,M),\bot), (B,(m',M'),\bot), \bot)$.

-Each relationship $rt_i = (rname_i, (A,(m_i,M_i),r_i), \ (B,(m_i',M_i'),r_i'), A_i) \in conn(A,B)$ contributes to the aggregated multiplicities defined as:

$$m := \underset{rt_i \in conn(A,B)}{\text{emax}} m_i, \qquad m' := \underset{rt_i \in conn(A,B)}{\text{emax}} m_i',$$
$$M := \bigoplus_{rt_i \in conn(A,B)} M_i, \qquad M' := \bigoplus_{rt_i \in conn(A,B)} M_i'.$$

-The boolean function *isGeneric*(*A*, *B*, *q*) tests whether the connection between *A* and *B* in query *q* is generic (i.e., non-typed). It is defined as:

$$isGeneric(A,B,q) \iff \big(|conn(A,B)| \geq 2\big) \ \wedge \ \big(\exists (x\!:\!A) - [z] - (y\!:\!B) \in match(q) \text{ such that } z \text{ is generic}\big).$$

NB: *e* is generic, which means that the query *q* does not specify a particular relationship type but considers all possible relationships between *A* and *B* (e.g., using the syntax–`[]`– or–`[v:]`–>).

In the GLDM of the running example (see Figure 10), there are two relationship types between the node labels: client and order. We can have functional requirements that are interested in checking the existence of a relationship between the client and the order. See, for example, the query Q1 in Listing: “Return all orders in which a client was involved in any role’. Rule 1 stipulates adding a generic relationship type between client and order node labels. Figure 18 illustrates this rule using our running example.

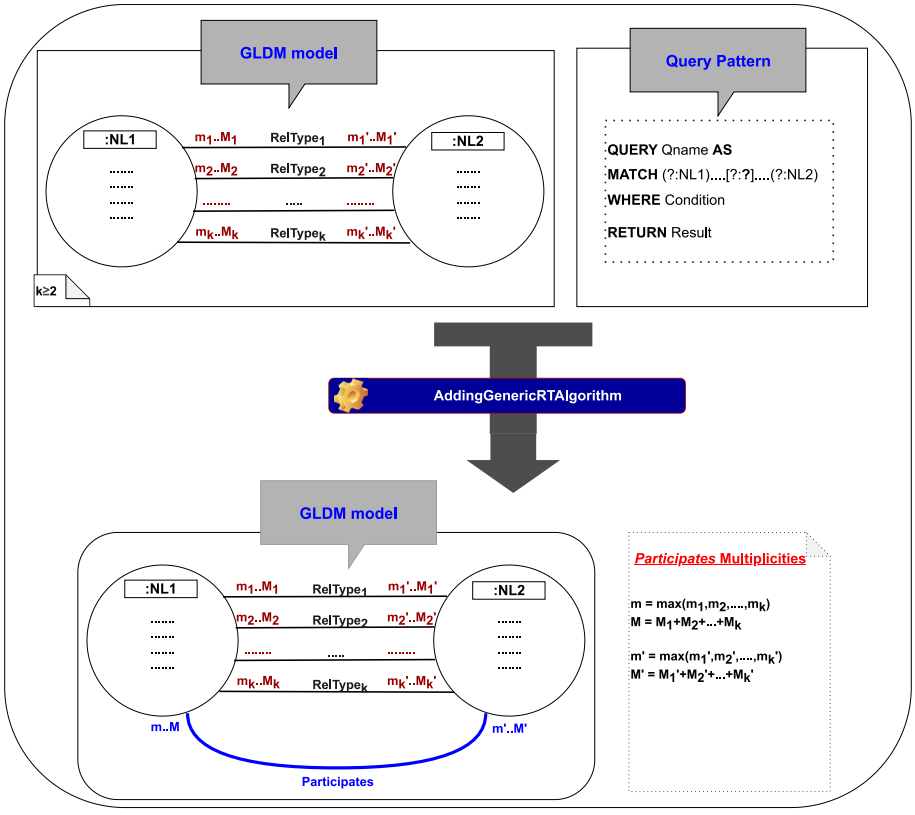


Fig. 17 Rule for Adding Generic Relationship Types

The generic relationship *Participates* is introduced to capture the fact that a client and an order may be connected through multiple relationship types appearing in the query workload, without explicitly specifying a particular one. On the *Client* side, two relationship types connect `Client` to `Order`: `MAKES` with multiplicity [1..1] and `PAYS` with multiplicity [0..1]. According to the constraint rule, the lower bound is computed using the *emax* function, which selects the maximum of the lower bounds of the involved relationships, yielding $\mathrm{emax}(1,0) = 1$. The upper bound is computed using the *extended sum* of the upper bounds, yielding $1 + 1 = 2$. Consequently, the multiplicity on the Client side of *Participates* is [1..2]. On the *Order* side, both `MAKES` and `PAYS` allow an order to participate in an unbounded number of such relationships, with multiplicities $[0..*]$. Applying *emax* yields a lower bound of 0, while *extended sum* preserves the unbounded upper bound, resulting in the multiplicity $[0..*]$.

Adding new node labels

Occasionally, we may add an intermediate node label instead of the relationship type. In the running example, the relationship type that links the product and seller node labels has two properties: the quantity in stock and the warranty. It may be difficult to pose queries involving these properties because the query processing time may be too long (since finding a specific warranty requires filtering on the product, finding the specific relationship (i.e., the edge) to the seller node, and then searching for the warranty). For example, the query Q2 in Listing: "return Products of a Seller, in which the warranty exceeds six months" requires the system to find the node of the given seller, then the

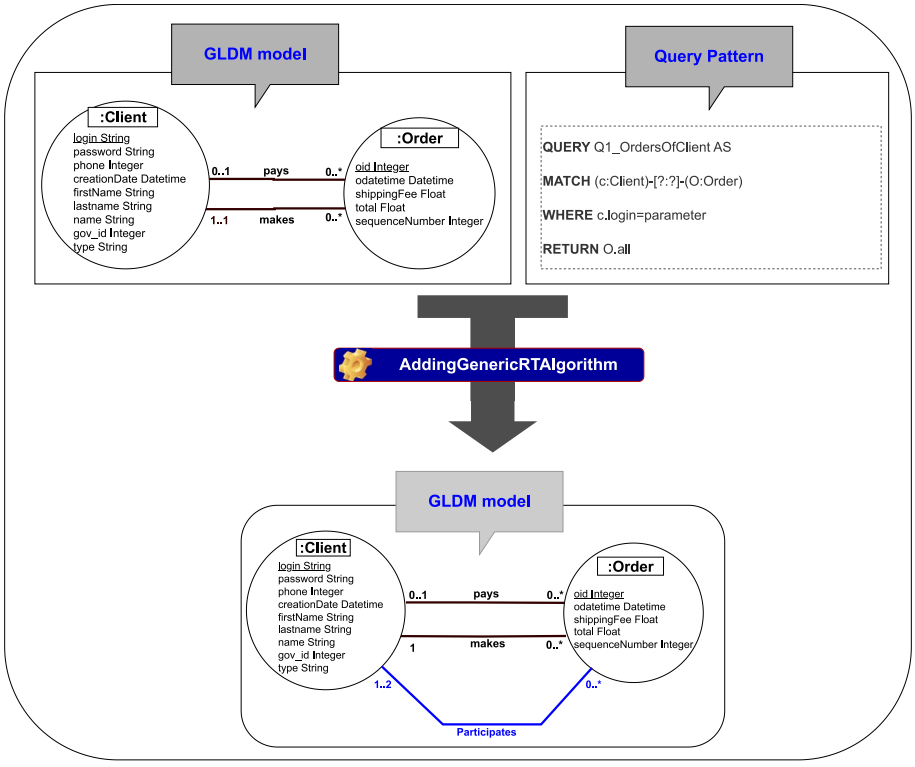


Fig. 18 Example of adding a generic relationship

edge from the seller node to their product nodes, afterward filter by the warranty property on the mentioned edge, and finally return the products that satisfy the condition.

Rule2: *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,* as presented in Figure 19 and illustrated by the example of the Figure 20.

Rule 2 is formally defined as follows:

$$\exists q \in Q,\ \exists rt_k \in RT,\ rt_k = (rtname, (A, (l_1, u_1), r_1), (B, (l_2, u_2), r_2), P_k),$$

such that: $(rtname \in M(q)) \ \wedge \ (\exists p \in P_k \text{ such that } p \in WR(q)) \implies$

1. **Creation of a new node label**

- $NL \leftarrow NL \cup \{nl_{new}\}$,
- $label(nl_{new}) \leftarrow name(p)$,
- $properties(nl_{new}) \leftarrow \{p\}$,

$properties(rt_k) \leftarrow properties(rt_k) \setminus \{p\}$.2. **Addition of two new relationship types:**

- $rt_1 = (p.name[NONSPACE]\&A.name, (nl_{new}, (?, ?), \perp), (A, (l_1, u_1), \perp), \perp)$,
- $rt_2 = (p.name[NONSPACE]\&B.name, (nl_{new}, (?, ?), \perp), (B, (l_2, u_2), \perp), \perp)$,
- $RT \leftarrow RT \cup \{rt_1\} \cup \{rt_2\}$.

A similar situation occurs when a filter is applied to a property that has a limited number of values, such as the type of client (namely, a company or person). According to best

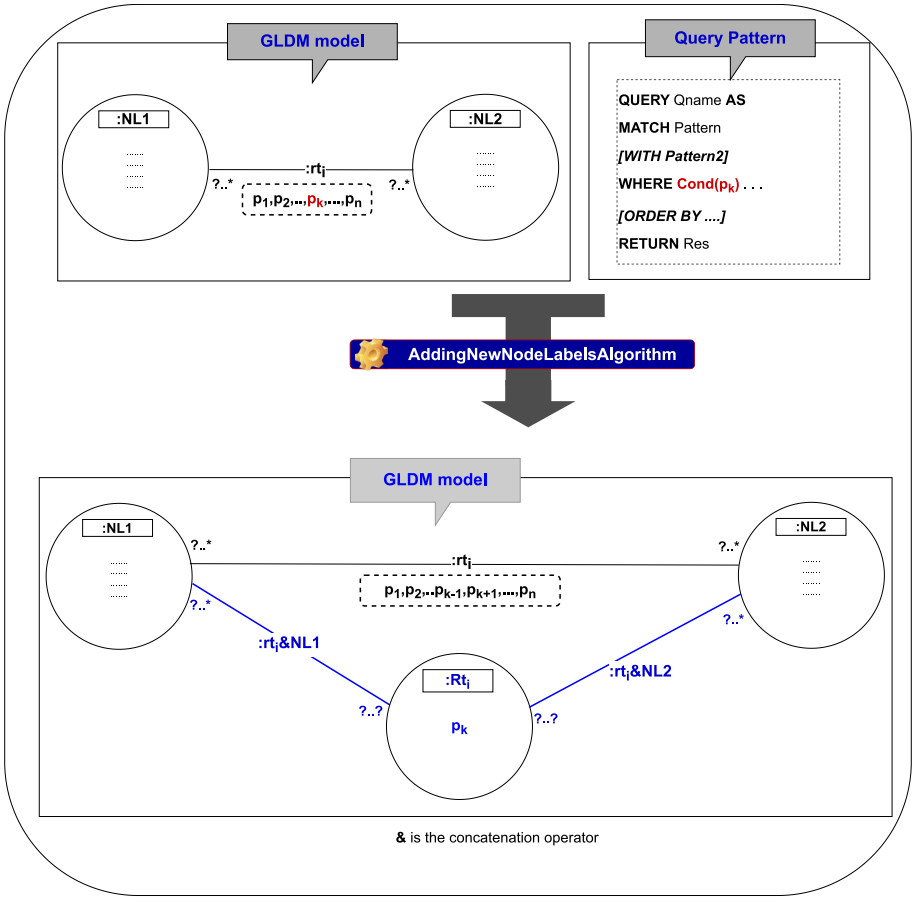


Fig. 19 Rule for adding new node labels using Rule 2

practices, if a query includes such a filter, it may be more efficient to define this property as a node label rather than to query and filter on the property value.

Rule3: *If there are queries that require filtering on a limited set of property values, this property should be transformed into a node label. The label of the added node will be the property's name, while its instances will be its possible values.* Figure 21 clearly illustrates this rule. As an example, for the query Q3 in Listing: "Company-type clients making orders on a given date", a new node labeled "typeOfClient" would be created in the GLDM (see Figure 22).

Rule 3 can be formally defined as follows (x is a parameter provided by the designer):

$$\exists q \in Q,\ \exists nl_k \in NL,\ nl_k = (label_k, P_k), \text{such that:}$$
$$(nl_k \in M(q)) \ \wedge \ \left(\exists p \in P_k \text{ such that } p \in W(q) \wedge |vals(p)| < x\right) \implies$$

1. **Creation of a new node label:**

- $NL \leftarrow NL \cup \{nl_{new}\}$,
- $label(nl_{new}) \leftarrow name(p)$,
- $properties(nl_{new}) \leftarrow \{p\}$,

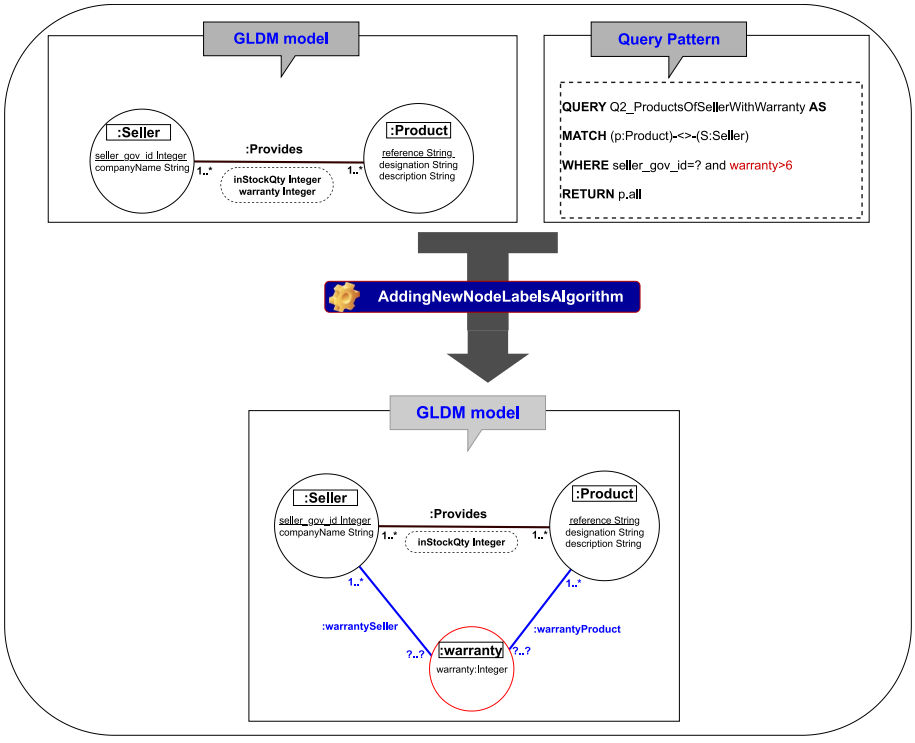


Fig. 20 Example of adding new node labels using Rule 2

$properties(nl_k) \leftarrow properties(nl_k) \setminus \{p\}$.2. **Addition of a new relationship type:**

- $rt_{new} = (\, name(p)\&\text{"of"}\&nl_k),\ (nl_{new}, (1,1), \perp),\ (nl_k, (?,?), \perp),\ \perp\,)$
- $RT \leftarrow RT \cup \{rt_{new}\}$.

Adding sequence relationship types

In the process of modeling, we often tend to ignore sequences. For example, the study case contains information about the client's orders: The model ignores the fact that a client's orders follow a sequence. Such a sequence may be referenced in queries. For example: "From a given order of a client, retrieve the next order". In this scenario, we establish a "next" unary edge within the Order node label.

Rule4: An edge should link instances of a node that are considered a sequence, creating a linked list. The direction of the link will depend on the queries (forward, backward, or bi-directional). By default, the schema will only add a forward edge if no such query exists.

For example, the query Q4 in the listing will return the next order of a given order for a given client. Figure 23 illustrates how to add a next or previous relationship next to the GLDM, while Figure 24 shows a use case for such rules.

Formally: $\exists q \in Q,\ \exists (x_1 : NL_1), (x_2 : NL_2) \in M(q),\ \exists (x'_1 : NL_1), (x'_2 : NL_2) \in WT(q)$

such that:$|conn(NL_1, NL_2)| > 0\ \wedge\ (id(x_1) = P_1 \in WR(q))\ \wedge\ (id(x_2) = P_2 \in WR(q))$
$\wedge\ (x_1.id = x'_1.id \in WR(q))\ \wedge\ (x'_2.p = x_2.p + 1 \in WR(q))$
$\Longrightarrow$
$RT \leftarrow RT \cup \{rt_{new}\}$,

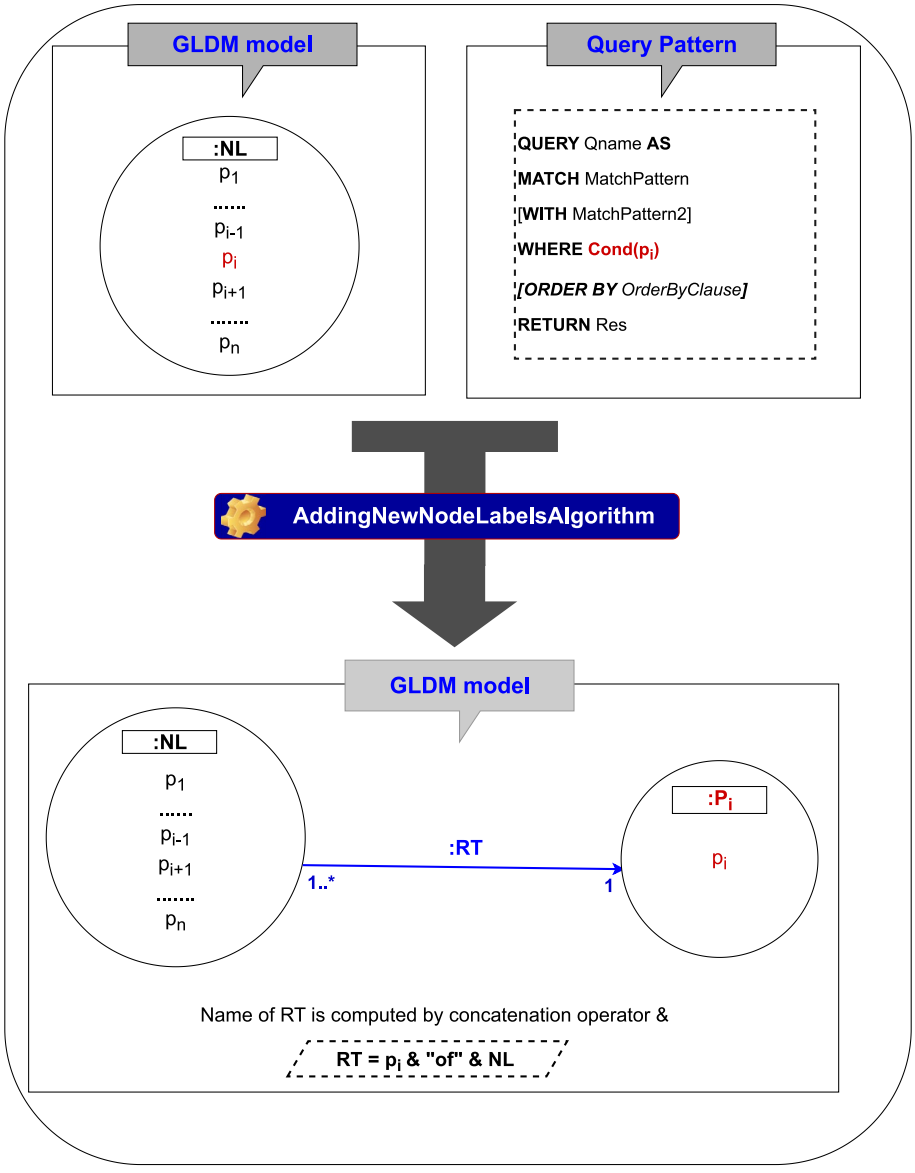


Fig. 21 Adding new node labels using Rule 3

where the new relationship type is defined as:

$$rt_{new} = (\text{“}next\text{”}, (NL_1, (0,1), \perp), (NL_2, (0,1), \perp), 2).$$

The above rule is defined for successor-based conditions, leading to the creation of the relationship type *“next”*. Symmetrically, when the query condition involves a predecessor relation, i.e., $x_2.p = x_2'.p - 1$, the same rule applies and produces a relationship type named *“previous”* whose direction is 2 (from NL_1 to NL_2)

Adding implicit relationship types

Sometimes queries may discover implicit relationships between node labels that are not present in the model. For example, we could store the product categories that a client has purchased. For example, the query Q5:

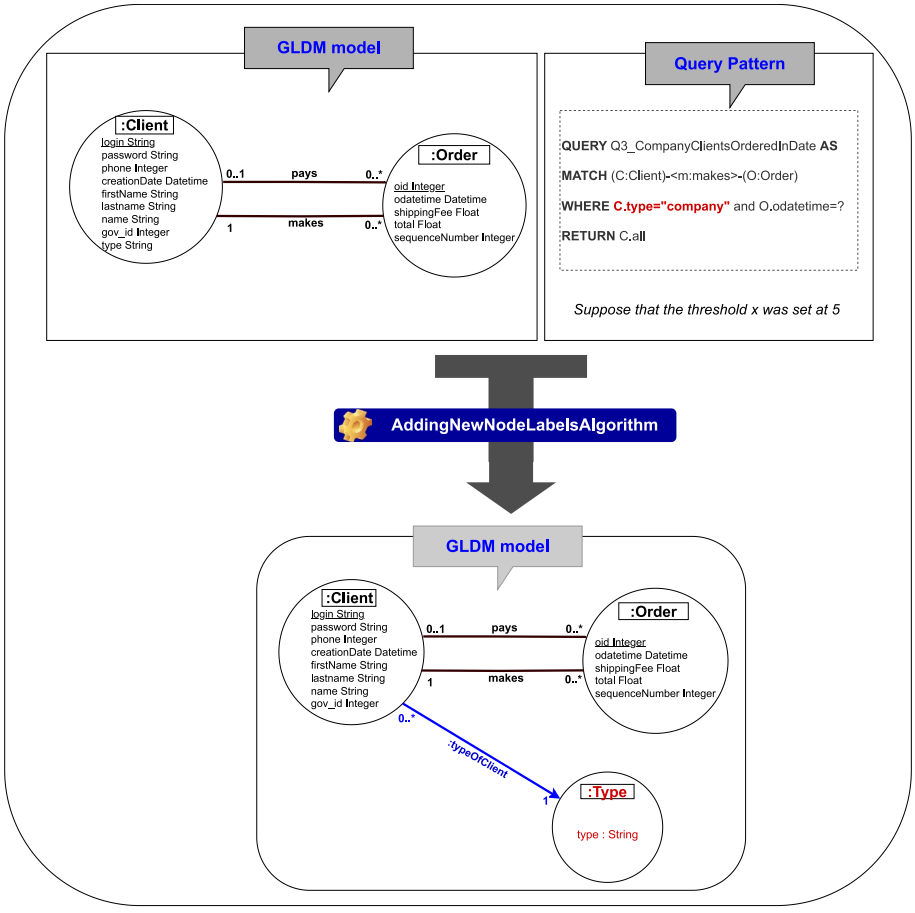


Fig. 22 Example of adding new node labels using Rule 3

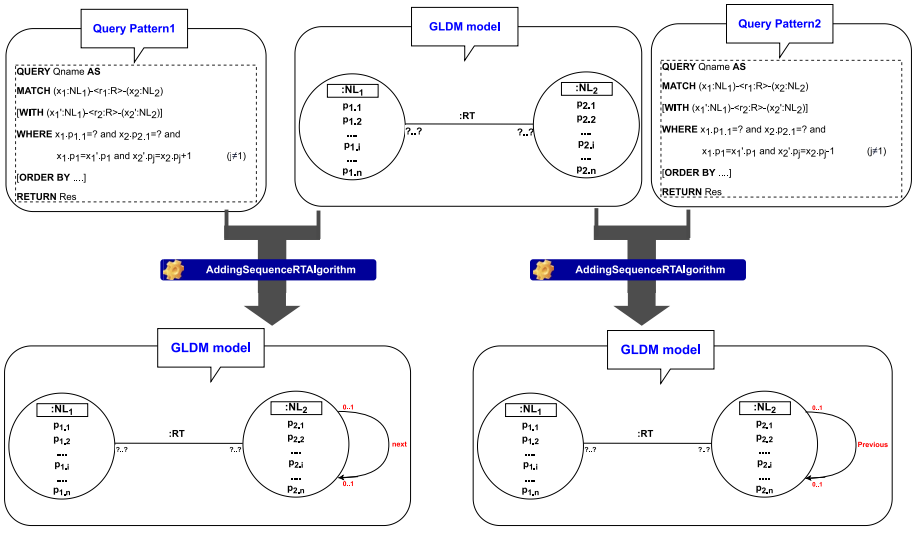


Fig. 23 Adding the sequence relationship type

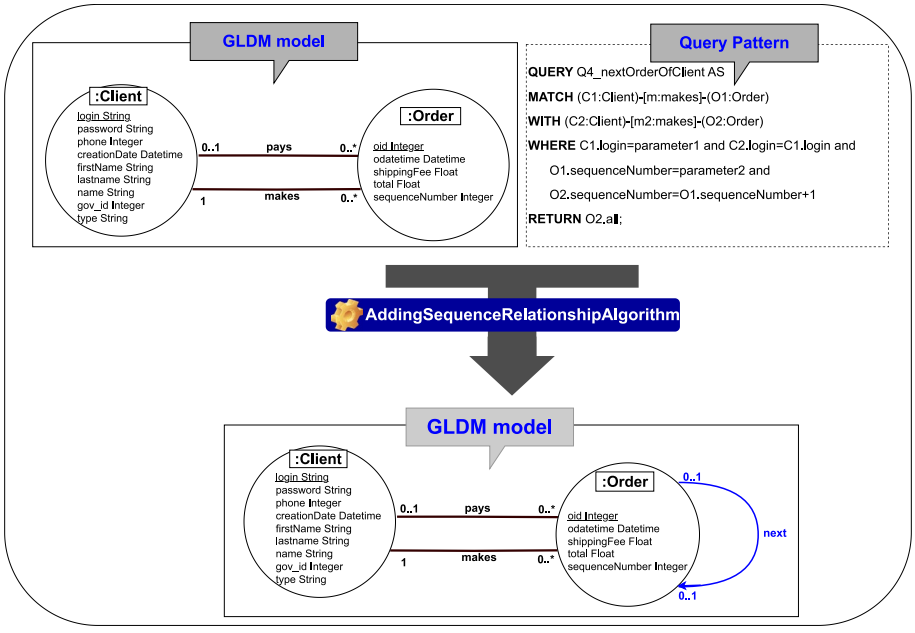


Fig. 24 Example of adding sequence relationship type

Match (c:Client)-[m:makes]-(o:Order)-[x:?]-(ctn:Contains)-[y:?]-(p:Product)-[z:?]-(ctg:category) Where c.login=? Return ctg.name

retrieves the product category names purchased by a client, knowing its login. The model does not explicitly explain this knowledge; it is implicit. In such a case, an explicit relationship type should be defined between the relevant node labels (namely, client and category node labels).

Rule5: Paths in the queries with implicit meaning should be changed to explicit relationship types (see Figure 25). This rule will add a new relationship type in the GLDM named *CategoriesPurchasedByClient* between client and category node labels, as shown in Figure 26.

Finally, the application of the different rules (from 1 to 5) will produce the query-driven GLDM in Figure 27.

Let $E = Nodelabs(M(q))$ denotes the ordered sequence of node labels involved in the graph pattern of query q and $n = |E|$. This rule is Formally defined as follows:

$$\begin{aligned}
&\exists q \in Q,\ |E| > 2,\ conn(x, y) = \varnothing,\ id(x) \in WR(q),\\
&\text{where } x = E[1] \text{ and } y = E[n]\\
&\quad \Longrightarrow\\
&RT \leftarrow RT \cup \{rt_{new}\}.\\
&\text{where } rt_{new} = (rtname, (x, (l_1, u_1), \perp), (y, (l_2, u_2), \perp), 2),\\
&rtname = name(q),\ \text{and } rt_{new} \text{ is directed from } x \text{ to } y \text{ (i.e., direction } = 2).
\end{aligned}$$

The multiplicities are obtained using the extended product along the path between x and y:

$$l_1 = \bigotimes_{i=1}^{n} l_{1i}, \quad l_2 = \bigotimes_{i=1}^{n} l_{2i}, \quad u_1 = \bigotimes_{i=1}^{n} u_{1i}, \quad u_2 = \bigotimes_{i=1}^{n} u_{2i},$$

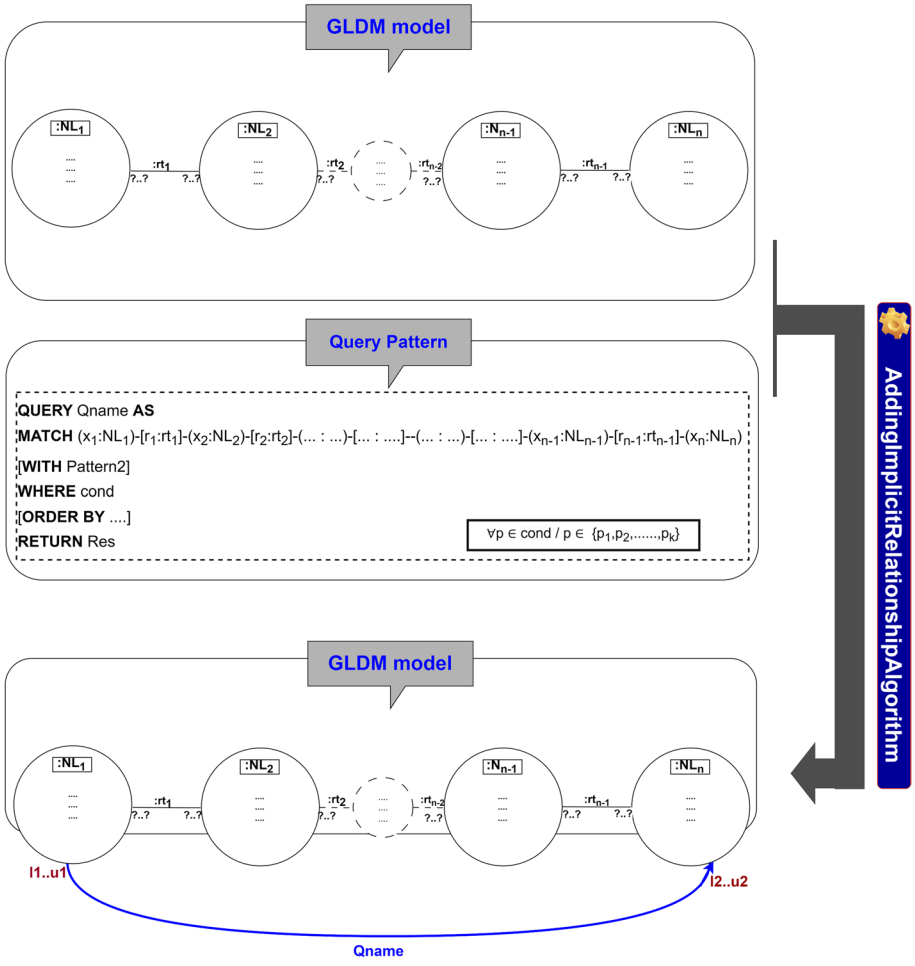


Fig. 25 Adding the implicit relationship type

where (l_{1i}, u_{1i}) and (l_{2i}, u_{2i}) are the multiplicity bounds associated with the related extremities of each intermediate relationship along the path between x and y.

For the new relationship type *CategoriesPurchasedByClient*, the multiplicities on both extremities are derived by applying the extended product along the query path between `Client` and `Category`. On the `Client` side, the path involves the relationships `MAKES`, `CONTAINS_Order`, `CONTAINS_Product`, and `is_composed_of`.

The lower bounds associated with these relationships include at least one 0, which, according to the definition of the extended product, yields a lower bound of 0. For the upper bounds, at least one intermediate relationship admits an unbounded maximum ($*$), and the extended product therefore propagates this unboundedness, resulting in an upper bound of $*$. Consequently, the multiplicity on the `Client` side is $[0..*]$.

Similarly, on the `Category` side, the extended product is applied to the multiplicity bounds of the same sequence of intermediate relationships, leading again to a lower bound of 0 and an unbounded upper bound $*$. As a result, the implicit relationship *CategoriesPurchasedByClient* is assigned the multiplicity $[0..*]$ on both extremities, faithfully reflecting the semantics of the original query path.

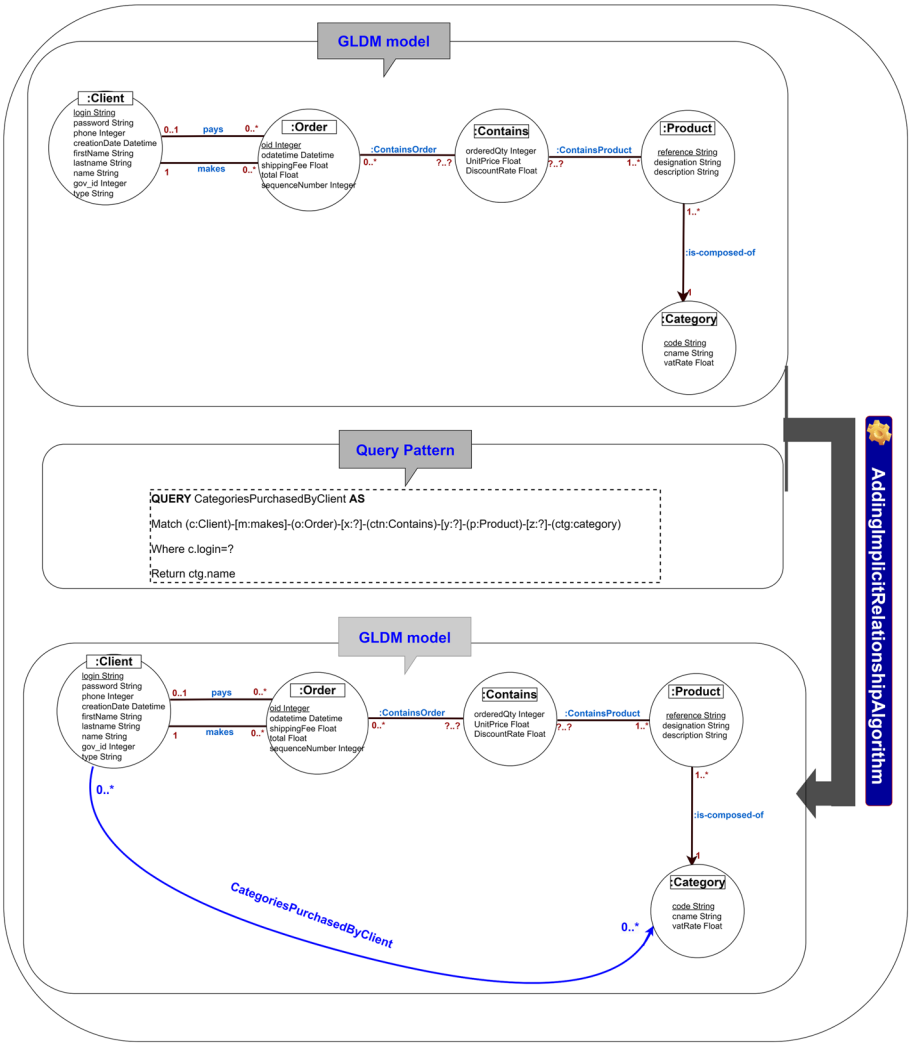


Fig. 26 Example of applying the implicit relationship types rule

Indexing the filtering or sorting properties

Indexes help optimize the process of finding specific nodes. So, without indexing, when querying a graph, the traversal process searches for nodes and relationships that meet our needs. By following relationships that match a specific graph pattern, we encounter elements that contribute to a query's result. However, in some situations, it is necessary to target specific nodes directly rather than discover them over the course of a traversal. Identifying the starting nodes for a traversal, for example, requires us to find one or more specific nodes based on some combination of labels and property values. To support efficient node lookup, graph-oriented DBMS allows for the creation of indexes per label and property. Similar to indexing node properties and labels, when queries involve traversing relationships and the database needs to quickly locate relationships of a specific type between nodes, it is important to index the relationship types to support efficient querying. In this work, property indexing is the only type studied. The properties involved in the WHERE clauses of the query model are the most indexed. Additionally, it would be useful to index the properties that contribute to the sorting criteria. Indexing can contain one or many properties. Therefore, index statements will enrich

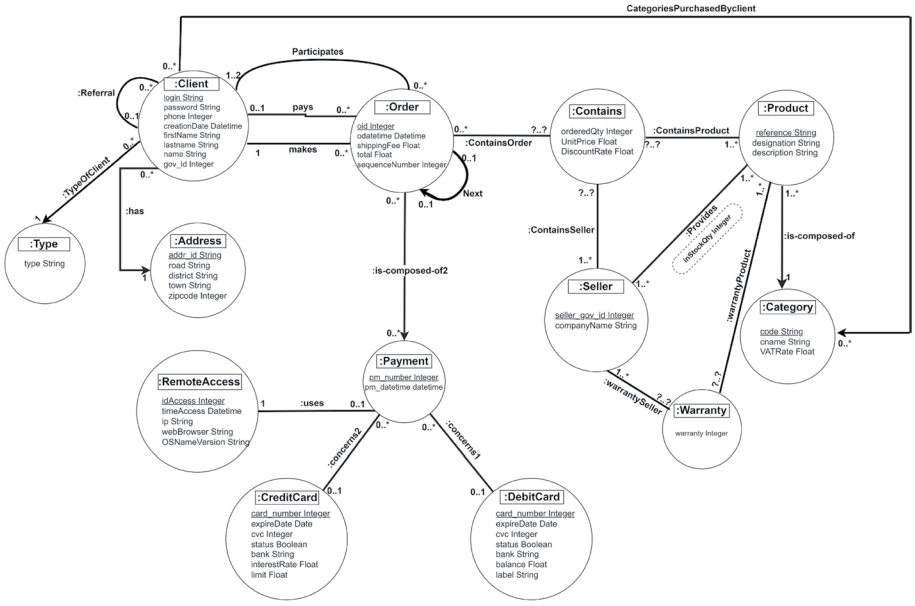


Fig. 27 GLDM after applying Rules 1–5

the GLDM. The statement $idxindex_name\{p_1, p_2, ..., p_k\}$ is an index statement named *index_name* having the properties $p_1, p_2, ..., p_k$ on which the index is composed. Some, but not all, graph databases implicitly index their properties with unique constraints; if a unique-type property required indexing, the statement would explicitly write it.

To add the indexing feature to the GLDM, we will use the query listing from the running example in the following. Table 3 provides instructions on how to add indexes to the final GLDM. Figure 28 displays the outcome of these rules.

In our methodology, we chose not to introduce a dedicated subsection for the physical level, since the only physical concern explicitly addressed is indexing. Although indexing is a physical mechanism, this concept alone does not constitute a sufficiently broad axis to justify a separate part. Therefore, we incorporated it into the logical level, and this integration is consistent with the fact that most, if not all, graph DBMS natively support indexing mechanisms.

Results and discussion

In this section, we showcase the significance of the methodology for graph-oriented database design, as outlined in Section "Result and discussion". We assess the solution's efficiency based on the time complexity of the various queries, and then we present their execution duration in a real hardware setup.

Query-based design and execution time

We will develop the implications of the various rules to demonstrate their importance in accelerating access queries.

Rule1:adding generic relationship types

Let X and Y be two node labels of sizes n and m, respectively.

Let RT=$\{RT_1, RT_2, ...RT_k\}$ be the set of relationship types between X and Y.

Let Q be a query that retrieves the nodes of X connected to nodes of Y with any relationship type $RT_1, RT_2, ..., or RT_k$.

Table 3 Properties indexing process

Query	WHERE properties	ORDERBY	Idx statements
Q1	login	—	idx idx1 login
Q2	warranty	—	idx idx2 warranty
Q3	type, odatetime	—	idx idx3 type; idx idx4 odatetime
Q4	login, oid, sequenceNumber	—	idx idx5 oid, sequenceNumber
Q5	login	—	—
Q6	label, status	—	idx idx6 label, status
Q7	label, pm_datetime	pm_datetime	idx idx7 label; idx idx8 pm_datetime
Q8	label, bank	—	idx idx8 label, bank
Q9	town	—	idx idx9 town
Q10	total, odatetime	—	idx idx10 total, odatetime
Q11	login	—	—
Q12	login,ip	ip	idx idx11 ip
Q13	login, card_number (credit card), oid	—	idx idx12 card_number (credit card)
Q14	code	—	idx idx13 code
Q15	card_number (debit card)	—	idx idx14 card_number (debit_card)
Q16	reference,seller_gov_id	—	idx idx15 reference,seller_gov_id
Q17	login	—	—
Q18	reference,seller_gov_id	—	—
Q19	odatetime	—	—

Using a data-driven design, the query Q has a statement of the following form:
MATCH (x:X)-[:?]-(y:Y) WHERE C RETURN R
Evaluating Q is the same as evaluating the following query:
MATCH (x:X)-[:RT_1]-(y:Y) WHERE C RETURN R
UNION
MATCH (x:X)-[:RT_2]-(y:Y) WHERE C RETURN R
UNION
....
UNION
MATCH (x:X)-[:RT_k]-(y:Y) WHERE C RETURN R

If the query starts from X nodes to Y nodes, it first matches Y nodes connected to X nodes using relationship type RT_1, then matches Y nodes connected to X nodes using relationship type RT_2,..., and finally matches Y nodes connected to X nodes using relationship type RT_k. In the end, the DBMS combines the results using the UNION operator in order to retrieve the X nodes connected to the Y nodes, regardless of the type of relationship.

If the GLDM were created using a query-driven design, a relationship type named Participates would be previously defined between X and Y nodes. Therefore, we will express query Q using the following statement:

MATCH (x:X)-[:Participates]-(y:Y) WHERE C RETURN R

The *participates* relationship might reduce the number of pattern matches required, potentially improving query performance. However, storing an additional relationship type (participates) adds some space complexity. Each relationship requires storage space for the relationship ID, type, and pointers to the connected nodes. This cost is usually minimal compared to the benefits of querying simplicity and query complexity reduction.

Rules 2–3:adding new node labels

This rule is called if a filter is applied to a property of a relationship type or to a node property with a limited set of values.

Transforming relationship properties into node labels can provide several benefits, particularly when it comes to filtering and querying those properties. Most graph-oriented DBMS support indexing on node properties but not on relationship properties. By transforming relationship properties into node properties, database stakeholders can create indexes on these new nodes, which significantly speeds up queries that filter based on these properties. In terms of path traversal efficiency, filtering on node properties can be more efficient than filtering on relationship properties, as many graph traversal algorithms are optimized for node properties.

Node labels represent node property values, making queries more straightforward and efficient. Instead of filtering nodes based on a property, the query can directly access the relevant nodes, avoiding the filtering costs. Thus, this rule leads to a more efficient query execution plan with reduced time complexity, especially for large datasets.

Rule4:adding sequence relationships By creating a sequence relationship between nodes, queries that need to retrieve the previous or next node can do so in constant time, O(1), by following the edge directly. This avoids the need to scan through all nodes to find the previous or next one. Thus, the database can directly traverse the "next"

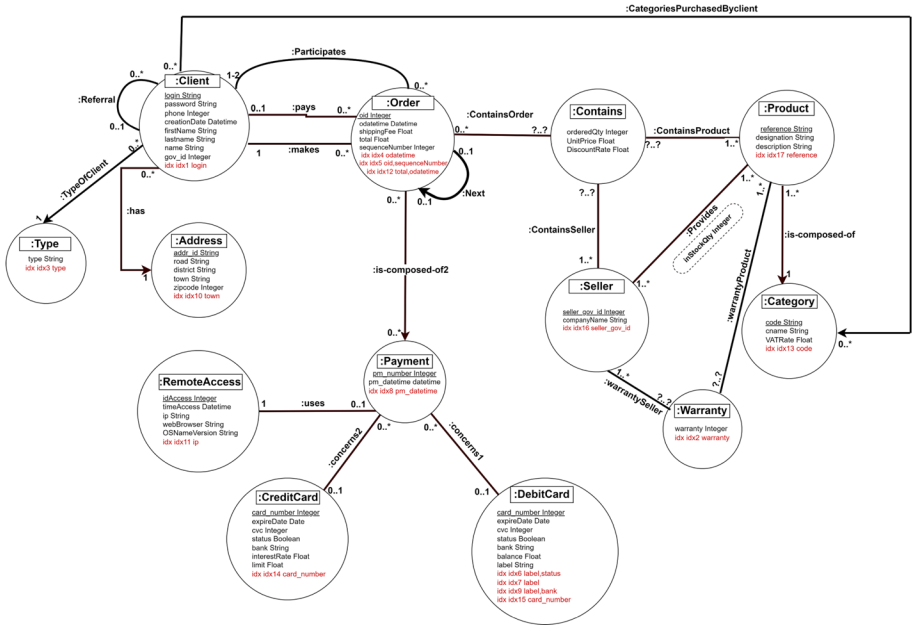


Fig. 28 GLDM after indexing

relationship without additional filtering or searching. Regardless of the size of the dataset, we can retrieve the next or previous item in a sequence.

Rule5:adding implicit relationships By adding explicit relationships, queries can directly traverse these relationships without needing to navigate through multiple intermediary nodes. This reduces the length of the traversal path and improves query response time. Therefore, direct relationships allow queries to access connected nodes in constant time, O(1), as opposed to O(n), or worse, which would be necessary if multiple jumps were required to compute the relationship.

Implementation

We have implemented a prototype to operationalize our pipeline from class diagrams to adjusted class diagrams and then to graph logical data models. The whole artefact set is published in a GitHub repository under a free license [30]. The repository contains the metamodels in both Ecore and Xcore: (i) the class diagram metamodel, (ii) the adjusted class diagram metamodel, and (iii) the graph logical data metamodel GLD2M. Alongside these, we also include the query metamodel (part of our DQR2M), plus instances of these metamodels used as running examples and for tests.

The transformation process combines M2M and M2T specifications. For M2M, we use ATL where mappings are structural and rule-driven: (i) class diagram → adjusted class diagram and (ii) adjusted class diagram → GLDM. When workload information must be analyzed globally–i.e., to derive a query-driven GLDM from the pair (GLDM, queries)–we employ Xtend, which offers imperative control, rich collections, and multi-pass analysis. This choice is appropriate when decisions depend on observing all elements rather than transforming elements one by one. For M2T, we rely on Xtend template strings to generate deployable artifacts (e.g., Neo4j Cypher for schema constraints, validation queries, and load scripts) query-driven GLD2M instances.

To make the approach usable, we provide a textual DSL built with Xtext for editing models and queries. The Xtext editor offers syntax highlighting, validation (aligned with the Ecore/Xcore metamodels), and quick fixes: editing a DSL file produces its instance model and can trigger the ATL/Xtend transformations and code generation. Finally, an examples/project bundle includes samples: DSL sources, their serialized EMF models, the intermediate adjusted/graph models, and the generated Cypher–so the entire process can be reproduced.

Experimental results

To quantify the impact of the different rules as well as the benefits of property indexing, we estimate the execution durations for different queries based on various scenarios. We experimentally verified the results using a workstation with Intel Xeon(R) Silver 4214R CPU @ 2.40GHz, 12 cores, 24 logical processors, and 128 GB of RAM running Windows 11. The system only runs system programs, the Python IDE, and the Neo4j DBMS (Edition Enterprise) [31] during the experimentation. We developed a Python program to randomly generate and insert data into five distinct graph databases derived from the same e-commerce running example, each serving a different modeling objective. The program is accessible in an external repository [29]. Two databases are named BaseLineDB and GraphDesignDB which are related to the baseline and the indexed query-driven GLDM of our article. Three other databases called 4VDesignDB, Uml2LpgDB, and extendedGdbsDB related to databases designed using selected approaches [7, 10, 11].

- **BaseLineDB.** It represents the baseline graph model and weighs approximately 1.74 GB. This value serves as a reference for assessing the additional storage costs introduced by more advanced design strategies. This database is a realization of the data-driven GLDM presented in Figure 10. Their nodes and edges respect the structure of this model.
- **4VDesignDB.** The schema of this database slightly increases the storage size to 1.76 GB. This overhead is mainly due to the explicit modeling of type hierarchies through additional nodes and relationships. In particular, subclasses of the Client class (e.g., company vs. individual) and subclasses of bank cards (credit vs. debit) are materialized as separate nodes, with dedicated edges linking each instance to its corresponding type node.
- **Uml2LpgDB.** The UML2LPG design further increases the database size to 1.91 GB. This additional cost is explained by the introduction of structural relationships aimed at supporting specific access patterns. In particular, consecutive orders of a client are explicitly linked using a sequencing relationship, and implicit relationships between clients and the product categories they purchased are made explicit through additional edges.
- **extendedGDB.** The Extended GDBS schema reaches 2.07 GB, reflecting the cumulative effect of multiple structural optimization rules introduced to accelerate query processing. These optimizations include additional relationships and reified structures that reduce traversal depth and filtering cost for some queries.
- **GraphDesignDB.** GRAPH-DESIGN exhibits the largest storage footprint, with a total size of 2.81 GB. This increase is primarily due to the combined effect of query-driven structural optimizations and systematic index creation on frequently accessed

properties (Figure 28). The additional nodes, relationships, and index structures significantly enlarge the physical storage size; however, this overhead is intentionally accepted as a trade-off to achieve substantial reductions in query execution time, especially for queries executed repeatedly at scale.

These databases are composed of **1000 000** clients, **2000 000** orders, **700 000** addresses, **100 000** product lines (i.e., contains nodes), **10 000** products, **5 000** sellers, **1500 000** payments, **600 000** bank cards, and **3000 000** remote accesses. The queries Q1, Q2, Q3, Q4, and Q5 took different execution times to run in this setup (Table 4). In total, the ecommerce databases designed independently of query workloads manage more than 9 million nodes and some millions of edges. The remaining databases hold additional nodes and relationships due to the various rules introduced to refine the schema.

All databases are stored on disk, as Neo4j uses persistent storage to maintain the graph structure, node and relationship records, property values, and index trees. The observed size increases are therefore attributed to the added cost of indexing structures and the additional elements introduced by the query-driven schema design.

Let us now analyze the observed results, query by query.

- **Query Q1: This query requires retrieving all orders in which a client is involved, regardless of the role.**

1. In the baseline schema the client-order relationship is split across multiple relationship types (e.g., MAKES, PAYS); the query engine must traverse all outgoing and incoming relationships from the Client node; no structural shortcut exists to aggregate client participation. As a result, the query explores a large neighborhood of the client node, leading to the highest execution time (32.56 s). This value serves as the reference.
2. 4VDesign introduces additional typing nodes (e.g., client subtypes, card subtypes), but does not modify the way orders are connected to clients. For Q1, the traversal pattern is identical to Baseline; the engine still explores the same client-order relationships; the extra nodes slightly increase graph density. Consequently, no performance improvement is observed, and there is even a negligible overhead due to the richer structure (32.68 s, +0.37%).
3. UML2LPG introduces sequencing links between orders and explicit relationships between clients and categories. However, none of these optimizations are relevant to Q1. The client–order access pattern remains unchanged, while the total number of edges in the graph increases. Result: same execution pattern as the baseline, with a minor overhead due to additional relationships (32.75 s, +0.58%).
4. Extended GDBS applies structural optimization rules that reduce traversal depth, simplify access paths, and aggregate frequently accessed semantics. For Q1, the schema introduces a more direct representation of client participation in orders. This significantly reduces the number of traversed relationships and the size of intermediate result sets. Result: execution time divided by more than 4, yielding a –76.6% reduction.
5. GraphDesign explicitly models client participation in orders using a query-driven relationship (e.g., Participates). Q1 is reduced to a single–hop traversal from Client to Order, without distinguishing roles at query time. This avoids scanning multiple

Table 4 Experimental results of execution duration for queries based on different strategies (in seconds)

Query	Basic	4VDesign	Uml2Lpg	Extended GDBS	GRAPH-DESIGN
Q1	32.56	32.68 (↗0.37%)	32.75 (↗0.58%)	7.62 (↘76.6%)	7.38 (↘77.4%)
Q2	0.87	0.88 (↗1.15%)	0.79 (↘9.2%)	0.75 (↘13.80%)	0.72 (↘17.25%)
Q3	206.85	34.26 (↘83.50%)	35.01 (↘83%)	52.44 (↘75%)	21.76 (↘89.50%)
Q4	254.31	262.77 (↗3.33%)	34.26 (↘86.53%)	34.86 (↘86.31%)	29.81 (↘88.28%)
Q5	1936.65	1932.98 (↘0.19%)	7.66 (↘99.60%)	8.01 (↘99.59%)	7.20 (↘99.63%)

relationship types and merging partial result sets. Result: best performance, with a −77.33% reduction compared to the baseline.

- **Query Q2: It retrieves the products of a given seller for which the warranty period exceeds six months.**

1. In BaseLine, the PROVIDES relationship directly connects Seller and Product, and warranty is a property on this relationship. The engine uses seller_gov_id to find the seller, then traverses all PROVIDES relationships from this seller. Therefore, it filters them with the predicate warranty>6, and finally, it returns the connected Product nodes. This gives the reference cost: scanning all outgoing PROVIDES edges of one seller and filtering on a relationship property.
2. 4VDesign enriches the schema with typing nodes for clients and bank cards but does not change the Seller-PROVIDES-Product pattern or where the warranty is stored. For Q2, the pattern is structurally identical to the baseline, and traversal is the same: Seller −[:PROVIDES]−> Product, filter on warranty. The extra typing nodes increase the overall graph density slightly without helping this specific query. This explains no improvement and a very small overhead (+1.15%).
3. In Uml2LPG, the seller-product association is reified as a `PROVIDES` node: sellers are linked to `PROVIDES` nodes via `SellerToProvides`, and these nodes are in turn linked to products via `ProductToProvides`. Attributes such as warranty and stock quantity are stored on the `PROVIDES` nodes themselves. Although this introduces an additional hop, it centralizes contract information in a single, homogeneous node label and enables early filtering on `PROVIDES.warranty`. This more regular structure explains the moderate improvement observed for Q2 (0.79s, −9.20%).
4. Extended GDBS further refines the schema with structural optimization rules that reduce the number of candidate relationships and intermediate results. The query planner would explore a smaller and more relevant subgraph, yielding an execution time of 0.75s (+13.79%).
5. GRAPH-DESIGN combines query-driven refinements with dedicated indexes on seller identifiers and warranty values, which allows the engine to discard non-qualifying data very early and attain the best performance for Q2 (0.72s, −17.24%).

- **Query Q3: This query filters clients by type (Company), then retrieves all orders made by those clients.**

1. In the Baseline schema, client type is stored as a simple property. Filtering on this property requires scanning a very large number of client nodes, then expanding MAKES relationships to order. With 1 000 000 clients and 2 000 000 orders, this leads to a massive candidate set, very large intermediate results, and therefore the worst execution time.
2. In 4VDesign, client typing is explicitly modeled. Client nodes are linked to either `Company` nodes or `Person` nodes via the relationships `IS_A_Client_Company` and `IS_A_Client_Person`. As a result, Query Q3 is transformed from filtering all clients by a property to traversing only clients connected to `Company` nodes. This reduces the number of candidate clients before expanding

`MAKES` relationships. Consequently, orders of company clients are isolated early, and the execution time drops to 34.26 s, corresponding to an 83.43% reduction compared to the baseline.

3. Extended GDBS adopts a similar approach for client typing, yielding comparable performance (35.01s, 83.07% reduction).
4. Uml2LPG applies several structural optimization rules, but because the client type is not modeled as a separate node, the set of candidate nodes remains larger, resulting in an execution time of 52.44s (a reduction of 74.64%).
5. In GRAPH-DESIGN, the schema explicitly models client types as dedicated `Type` nodes. As a result, Query Q3 is evaluated by starting directly from the `Company` type node and traversing only the clients that belong to this category before expanding the `MAKES` relationships toward orders. Combined with dedicated indexes on frequently accessed properties, this highly selective traversal led to the best performance for Query Q3, with an execution time of 21.76s, corresponding to an 89.48% reduction.

- **Query Q4: This query retrieves the next order of a given client based on order sequencing.**

1. In the BaseLine schema, order succession is not explicitly modeled, and sequence information is stored only as an attribute. As a result, the query must retrieve all orders of the client and evaluate the predicate `o2.sequenceNumber = o1.sequenceNumber + 1` at runtime, leading to very large intermediate result sets and a high execution time (254.31s).
2. 4VDesign does not introduce any structural refinement for order sequencing and therefore exhibits a similar access pattern, with even a slight overhead due to additional typing nodes (262.77s).
3. UML2LPG, Extended GDBS, and GRAPH-DESIGN all materialize order succession explicitly, thus avoiding property-based comparison on sequenceNumber. This structural choice yields an important reduction in execution time compared to the baseline and 4VDesign schemas. GRAPH-DESIGN goes further by introducing dedicated indexes (e.g., on Client.login, Order.sequenceNumber), and this explains why the execution time is lower in GRAPH-DESIGN.

- **Query Q5: Categories purchased by a client**

1. In baseline, Q5 requires traversing a long path: `Client` → `Order` → `CONTAINS` → `Product` → `Category`. The query explores a large number of orders, multiplies intermediate results at each hop, and produces extremely large traversal spaces. Result: very high execution time, the worst of all queries.
2. 4VDesign introduces typing nodes but does not modify the purchase-category access path. The traversal pattern for Q5 is identical to Baseline. This is why we have the same execution time as Baseline, confirming that typing alone does not help deep traversals.
3. UML2LPG explicitly introduces a direct relationship between clients and categories, materializing the implicit semantics "categories purchased by a client". Q5 is therefore transformed from a four-hop traversal into a short, direct path. UML2LPG introduces only order sequencing and this explicit client-category

relationship. So, the graph remains relatively less dense than in Extended GDBS and GraphDesign. As a consequence, traversal depth is significantly reduced (7.66 s, –99.60%).

4. Extended GDBS also introduces an explicit relationship between clients and categories, allowing Q5 to avoid deep traversal. However, Extended GDBS additionally introduces other structural relationships (e.g., sequencing, generic edges) to support a broader set of query patterns. This makes the graph richer but also denser, which slightly increases traversal overhead. As a result, the execution time remains extremely low (8.01 s), but marginally slower than UML2LPG due to higher graph density.
5. GraphDesign relies on the same structural principle, namely the explicit materialization of the client-category relationship, but lookup is faster and more selective due to dedicated indexes. As a result, we obtain the best observed execution time (7.20 s, i.e., a 99.63% reduction compared to the baseline).

The proposed design methodology focuses on optimizing query execution time by increasing the number of edges in the database while minimally affecting the number of nodes. With the proposed approach, the database now has 6.5 million edges (about 50% more edges), reflecting a significant rise in space complexity, while the number of nodes increases slightly (in total the indexed query-driven database required 63% more space in comparison with the basic version). This trade-off enables important improvements in query performance, as the additional edges effectively reduce the computational overhead of traversing the graph. As a result, the method demonstrates a good balance between space complexity and execution time, cutting execution time by more than 73% compared to the basic version and achieving a noticeable performance boost for query-intensive workloads.

Managerial implications

The findings of this study have significant managerial consequences for organizations seeking to optimize their database infrastructure. The results of the analysis propose several managerial recommendations that are discussed as follows:

- By structuring graph databases based on a query-driven model, managers can enhance data storage and retrieval efficiency, leading to improved performance in handling complex relationships.
- The methodology facilitates informed decision-making regarding IT investments, particularly in determining when and how to transition from relational to graph-based storage systems.
- The ability to execute real-time queries makes this approach highly valuable for industries such as e-commerce, finance, and cybersecurity, where fraud detection and customer behavior analysis are critical.
- By leveraging MDA, businesses can automate key aspects of the database design process, reducing manual effort and lowering development costs.

These benefits collectively contribute to the scalability, adaptability, and cost-effectiveness of graph database implementations in various business domains.

Conclusions

This work addressed GRAPHDESIGN, a robust graph data modeling methodology. We have presented the principles of graph database modeling for Neo4j and have defined transformation rules to move from the conceptual level to the logical and physical levels.

The present article has significant repercussions for researchers and practitioners. The literature review demonstrates that the design of the NoSQL database can be beneficial for understanding their data storage as well as standardizing their access. The use of MDA in conjunction with query-driven design rules brings several advantages. The MDA simplifies the modeling process's automation, whereas the query model's support is compatible with the best practices in graph-oriented database design. Moreover, the proposal reveals a collection of models at several levels. The goal would be to enrich, if necessary, the process, particularly at the logical and physical levels.

Despite its advantages and contributions, this work also has some limitations:

1. Firstly, the paper introduces metamodels that support different types of queries. However, the illustration and application only encompassed some read queries, which hold significant importance in a NoSQL context because of their execution times. As a perspective, we intend to extend our study to detail the different types of queries, including aggregation read and short path queries.
2. On another side, while indexing can significantly speed up query execution, especially for frequently used properties in WHERE and ORDER BY clauses, it is not always the best solution in every case. Every addition, modification, or deletion of data necessitates the updating of indexes, thereby impacting write operations. Moreover, all experiments were conducted using a single graph database management system and the evaluation relies on a mid-sized dataset. While this scale is sufficient to highlight the structural impact of schema design choices, it does not capture the full complexity of very large-scale industrial deployments. Write performance, update-intensive workloads, and distributed deployments are therefore left for future work.
3. When different queries require incompatible schema optimizations, the proposed methodology does not attempt to enforce all refinements simultaneously; instead, the selection of optimizations is currently performed manually, based on their expected global benefit and the importance of the targeted queries. To scale the approach to larger workloads, queries are intended to be grouped according to similar access patterns so that optimizations can be applied at the level of query clusters rather than individually; in a future work, we plan to formalize and partially automate these decision and clustering processes.
4. In addition, due to the increasingly importance role of constraints in graph-family DBMS, we plan to take into account the integrity constraints in a later academic work, which will contribute to the data integrity in complementarity with the design.
5. Finally, node multi-labeling can occur in some use cases; this is why this aspect also needs a particular interest, and thus it is therefore left for subsequent investigation.

Abbreviations

ACD	Adjusted class diagram
ACID	Atomicity, consistency, isolation, and durability
ATL	Atlas transformation language
CD	Class diagram
CIM	Computing-independent model
CRUD	Create, Read, Update, and Delete
DBMS	Database management system(s)

DQR2M	Data and query-related metamodel
DSL	Domain-specific language
EMF	Eclipse modeling framework
ERD	Entity relationship diagram
GLDM	Graph logical data model global positioning system
GRAPHDESIGN	Graph-oriented database design approach
LPG	Labeled property graph
M2M	Model-to-model
M2T	Model-to-text
MDA	Model-driven architecture
NoSQL	Not only SQL
PIM	Platform-independent model
OCL	Object constraint language
PSM	Platform-specific model
UML	Unified modeling language

Author contributions
AD conceptualization, data curation, writing - the original draft. MH methodology, formal analysis, writing - the original draft. HO software, methodology, project administration. EBT funding acquisition, investigation, visualization. DGA investigation, software, visualization. IA supervision, validation, writing - review and editing. All authors read and approved the final manuscript.

Funding
This study was funded by the European University of Atlantic.

Data availability
All materials used in this study are publicly available at the links https://github.com/dourhri84/gdbm https://github.com/dourhri84/graphDbDesigner.

Code availability
All materials used in this study are publicly available at the links https://github.com/dourhri84/gdbm and https://github.com/dourhri84/graphDbDesigner.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 1 May 2025 / Accepted: 29 March 2026
Published online: 03 May 2026

References

1. Akoka J, Comyn-Wattiau I, Laoufi N. Research on big data-a systematic mapping study. Comput Stand Interfaces. 2017;54:105–15.
2. Dourhri A, Hanine M, Ouahmane H. Kvmod-a novel approach to design key-value nosql databases. Inform. 2023;14(10):563.
3. DB-Engines: Knowledge Base of Relational and NoSQL DBMS. https://db-engines.com/en/ranking/graph+dbms
4. Atzeni P. Data modelling in the nosql world: A contradiction? In: Proceedings of the 17th International Conference on Computer Systems and Technologies 2016, 2016;1–4
5. Kaur K, Rani R. Modeling and querying data in nosql databases. In: 2013 IEEE International Conference on Big Data, 2013;1–7. IEEE
6. Roy-Hubara N, Rokach L, Shapira B, Shoval P. Modeling graph database schema. IT Prof. 2017;19(6):34–43.
7. Roy-Hubara N, Sturm A, Shoval P. Designing nosql databases based on multiple requirement views. Data Know Eng. 2023;145:102149.
8. Melouk M, Rhazali Y, Youssef H. An approach for transforming cim to pim up to psm in mda. Procedia Comput Sci. 2020;170:869–74.
9. Sprinkle J, Rumpe B, Vangheluwe H, Karsai G. Metamodelling: state of the art and research challenges. In: Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems, 2007;57–76.
10. Akoka J, Comyn-Wattiau I, Prat N. A four v's design approach of nosql graph databases. In: Advances in Conceptual Modeling: ER 2017 Workshops AHA, MoBiD, MREBA, OntoCom, and QMMQ, Valencia, Spain, November 6–9, 2017, Proceedings 2017;36, 58–68 . Springer
11. León A, Santos MY, García A, Casamayor JC, Pastor O. Model-to-model transformation from UML class diagrams to labeled property graphs. Bus Inf Syst Eng. 2024;66(1):85–110.
12. Iñiguez G, Battiston MKF. Bridging the gap between graphs and networks. Commun Phys. 2020;3:88.

13. Chebotko A, Kashlev A, Lu S. A big data modeling methodology for apache cassandra. In: 2015 IEEE International Congress on Big Data, 2015;238–245. IEEE
14. Poly Sil Sen NM. An ontology-based approach to designing a nosql database for semistructured and unstructured health data. Clust Comput. 2023;27:959–76.
15. Chen L, Davoudian A, Liu M. A workload-driven method for designing aggregate-oriented nosql databases. Data & Knowledge Engineering. 2022;142:102089.
16. Vega A, García-Saiz D, Blanco C, Zorrilla M, Sánchez P. Mortadelo: automatic generation of nosql stores from platform-independent data models. Future Gener Comput Syst. 2020;105:455–74.
17. Candel CJF, Ruiz DS, García-Molina JJ. A unified metamodel for nosql and relational databases. Inf Syst. 2022;104:101898.
18. De Virgilio R, Maccioni A, Torlone R. Model-driven design of graph databases. In: Conceptual Modeling: 33rd International Conference, ER 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings 33, 2014;172–185 . Springer
19. Abdelhedi F, Ait Brahim A, Atigui F, Zurfluh G. Mda-based approach for nosql databases modelling. In: Big Data Analytics and Knowledge Discovery: 19th International Conference, DaWaK 2017, Lyon, France, August 28–31, 2017, Proceedings 2017;19, pp. 88–102 . Springer
20. Sousa VM, Cura LMdV. Logical design of graph databases from an entity-relationship conceptual model. In: Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services, 2018;183–189
21. Erven WSSLHVMHR. Carvalho: designing graph databases with graphed. J Database Manag. 2019;30:41–60.
22. Angles R, Hogan A, Reutter JL. Pg-schema: schemas for property graphs. Proc VLDB Endow. 2023;16(11):2879–92.
23. Link S, Skavantzos N. Entity/relationship graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data 2025.
24. Daniel G, Sunyé G, Cabot J. Umltographdb: mapping conceptual schemas to graph databases. In: Conceptual Modeling: 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings 2016;35, pp. 430–444 . Springer
25. Davoudian A, Chen L, Liu M. A survey on nosql stores. ACM Comput Surveys (CSUR). 2018;51(2):1–43.
26. Ambler SW. The elements of UML (TM) 2.0 style. In: The Elements of UML (TM) 2.0 Style 2010. Cambridge University Press
27. Talluri S, Łuszczak A, Abad CL, Iosup A. Characterization of a big data storage workload in the cloud. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. ICPE '19, pp. 33–44. Association for Computing Machinery, New York, NY, USA 2019. https://doi.org/10.1145/3297663.3310302 .
28. al NF. Cypher: An evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data, 2018;1433–1445 . SIGMOD
29. Dourhri A, Hanine M, Ouahmane H. Github repository. https://github.com/dourhri84/gdbm
30. Dourhri A, Hanine M, Ouahmane H. Github repository. https://github.com/dourhri84/graphDbDesigner
31. Neo4j: Offcial website of Neo4j. https://neo4j.com/

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.