Conceptual data normalisation from the practical view of using graph databases

Vojtěch Merunka[1,2], Himesha Wijekoon[1], Pavel Beránek[3]

[1] Department of Information Engineering, Faculty of Economics and Management, Czech University of Life Sciences Prague, Prague, Czech Republic
(merunka|wijekoon)@pef.czu.cz
[2] Department of Software Engineering, Faculty of Nuclear Sciences and Engineering, Czech Technical University in Prague, Prague, Czech Republic
vojtech.merunka@fjfi.cvut.cz
[3] Department of Informatics, Faculty of Science, Jan Evangelista Purkyně University in Ústí nad Labem, Ústí nad Labem, Czech Republic
pavel.beranek@ujep.cz

Abstract. This article deals with a practical and synthetic view of conceptual modelling. It suggests four graph database normal forms organised into two levels of conceptual modelling: data and metadata, with room for yet one conceivable graph normal form based on old approaches, such as object-oriented class normalisation and the idea of conceptual symmetry. Attention is also paid to bridging the semantic gap between a database on the server side and a programming language on the client side, which argues for using graph databases as better data sources for business intelligence systems and working with machine learning language models. The authors applied their practical experience in teaching database modelling at a university and many years of experience in software development in Smalltalk, Python, Java, and C#.
Keywords. Graph database normal forms; graph databases; conceptual modelling; large language models; business intelligence.

1 Introduction

As stated in [3], software engineers and practitioners increasingly use graph databases. It is also evidenced by the activity FDIS 39075 [11] of the ISO/IEC consortium. The professional community agrees that these databases do not require such complex query programming for business intelligence applications, and their data model is more advantageous for machine learning large language models. These databases work differently because, unlike relational databases, they allow more helpful work with paths between individual data records; it is even possible to search for unknown paths in queries and work with the found paths as data objects. [16]

Robinson at al. in [19] say that graph databases offer a powerful platform for representing and analysing interconnected data entities, providing businesses with the tools to gain comprehensive insights into their data landscape and drive informed decision-making because they can capture complex relationships, graph databases are well-

suited for various business intelligence tasks, including relationship analysis, network analysis, real-time analytics, fraud detection, and risk management.

Moreover, graph databases can serve as the foundation for implementing dynamic knowledge graphs, as Chen and Xing [5] highlighted. This capability holds a good potential for supporting Large Language Model (LLM) based applications, as demonstrated by recent studies by Huang et al. in [10] and Zhou et. al in [20] which argue leveraging graph databases to build and query knowledge graphs, businesses can enhance the semantic understanding of their data, enabling more accurate and contextually rich language processing and generation within LLM frameworks.

However, followed by [9] and [14], these advantageous properties do not appear automatically and depend on the design. Therefore, software developers are confronted with the question, "How to design the best database structure?" In the past, thanks to the enthusiasm for the new technology of graph databases, it was spread that graph databases do not need many formal techniques and that only importing the data from the original relational tables into the new database engine was enough.

In this article, *four graph normal forms* will be shown, which the authors practically use in their work at the university, as well as software projects in practice. Their connections to a similar technique for designing object class structures by Scott W. Ambler [1] will be discussed as well.

2 Motivation

If a new idea is grounded in analogies or inspiration from various seemingly unrelated sources, it is not necessarily a negative; rather, it may serve as a positive indicator that we are on the right path of understanding. Indeed, this was the approach of ancient thinkers, and, regrettably, it is somewhat forgotten in today's era.

In the history of computer science, there has often been a tendency to forget and rediscover tools and methods under new names. In a way, *graph databases* serve as a prime example of this phenomenon, as they can be viewed as network databases from the 1960s, yet finely adapted to contemporary computer technology. Therefore, this is not a journey into the past but rather a significant advancement, where the original term "network database" might undermine its progress.

The concept of *symmetry* has a rich history. One of the earliest thinkers to systematically engage with symmetry was the ancient Greek philosopher Pythagoras. His school developed the concept of symmetry concerning geometry and harmony, believing it to be a fundamental principle in the physical world and cosmic order. Moreover, in the realm of the physical world, symmetry emerges as an observable mathematical attribute inherent to real systems, remaining invariant under specific transformations. Analogy operates as a mechanism whereby one subsystem's structural attributes are mapped onto another's, elucidating fundamental principles akin to those by which the divine constructs the universe, thus meriting our careful consideration. Drawing parallels to historical examples, such as the Russian chemist Dmitri Mendeleev's work in 1869 reveals symmetry and analogy's power in scientific endeavours by creation of the first widely recognised periodic table of chemical elements exemplifies this, as he ingeniously

organised known elements based on recurring properties while also predicting the existence and properties of yet-to-be-discovered elements. Remarkably, subsequent discoveries validated all of Mendeleev's predictions. Similarly, our aspiration for systematising graph database normalisation rules echoes Mendeleev's method, and we anticipate some benefit in enhancing the efficiency and effectiveness of the database designs.

3 Attempts to overcome the limitations of relational databases

In the past, several interesting object-oriented databases and the ODMG-93 standard [4] have also appeared and are still used in practice. So, it will be interesting to see if graph databases will evolve to support inheritance and composition in the future. Even in this case, today there is no single universal database programming and query language available as it is said in [16].

The survey [2] says that graph databases are not just slightly different from relational databases. Indeed, there are significant qualitative differences compared to relational databases that have great practical use, for example, in language models for machine learning or business intelligence systems.

For all reasons, getting serious about graph databases makes sense. Because graph databases are different from relational databases even on a conceptual level, it would be a mistake to import data structures from existing relational databases or design new structures using procedures from the design of relational databases. The various new and advantageous features would not be reflected, and we would have just the old relational database structure embedded in the graph database engine. However, there is no agreement on the conceptual design methods of a graph database, and we still need to have a generally accepted standard. However, we can be inspired by the procedures described later in this article.

4 Object class normalisation - Ambler's approach

Researchers have been interested in the normalisation of object-oriented structures since the early 1990s. Initially, this research emphasised improving relational techniques to be effectively used in object-oriented systems. With the advent of object-oriented databases, the focus has also moved towards object-oriented class normalisation. Object-oriented database normalisation was introduced as class normalisation by Ambler [1] inspired by Coad and Yourdon from [6]. Hence, Ambler has proposed notable initial ideas regarding object-oriented normalisation.

In a very recent paper, Lo et al. developed seven steps for object-oriented normalisation. [Lo] Their approach was based on both Ambler's class normalisation and relational database normalisation concepts. They took Ambler's approach until the third object-oriented normal form (3OONF) and came up with 4OONF, like the 4th relational database normal form. However, in contrast to Ambler's steps, they suggest generalisation to eliminate homogenous operations between classes. A similar approach to Ambler can also be found by Molhanec in [17].

There has been also few research to normalise object-oriented design not analogically to relational database normalisation. Falleri et al. have proposed a methodology to remove duplicate attributes by introducing general classes. [8] Their approach is based on relational concept analysis (RCA) and supports model-driven engineering (MDE) by automating the discovery of new classes and attributes when normalising. Ubaid et al. have developed the Class Hierarchy Normal Form Pattern (CHNFP) to maintain class schema in an object-oriented database. [15] CHNFP helps to manage objects and their network of objects in a memory-efficient manner by optimising the object graph loaded into the memory and controlling the inheritance hierarchy.

S. W. Ambler is also a pioneer of the agile approach in programming. He has published three object-oriented normal forms for object-oriented application development using agile methods. [1] These normal forms are like the first, second and third relational normal forms but use a different theoretical apparatus than relational normal forms. The relational normalisation is based on the functional dependencies between separate attributes, but Ambler's rules are based on various relationships of different subsets of attributes of objects. Ambler also talks about three object-oriented normal forms as a tool for class structure design complementary to the technique of design patterns.

5 Frisendal's approach to the graph normalisation

Frisendal's approach to the Graph Normal Forms [9] extends the classical relational normal forms without the distinction between the 3rd NF and Boyce-Codd NF and is expressed in a more "programmers-friendly" way but using the ISO/IEC 24707 standard [13]. The first five normal forms are more-or-less identical with the relational normal forms, and the 6th relational normal form is declared as the specific *Graph Normal Form*. The whole approach is based on two software engineering concepts, *primary key* and a *functional dependency*. Frisendal argues that functional dependencies, well known from relational database modelling, are also the basic construction element for graph database modelling if extended. Frisendal's (reformulated classical relational normal forms for the graph database modelling purpose) are and added a specific Graph Normal Form as follows:

- NF1: Eliminate Repeating Groups.
- NF2: Eliminate Redundant Data.
- NF3: Eliminate Columns Not Dependent on Key.
- NF4: Isolate Independent Multiple Relationships.
- GNF: Remaining Functional Dependencies Rule

GNF: Remaining Functional Dependencies Rule

This rule solves the identity and uniqueness of data in detail. Frisendal argues that identity and uniqueness are not the same concepts as it is in a relational database where there is no other concept of identity than (based on the old Codd's rules [7]) to use one or more column values as the unique identification of the whole record stored in a row

of the table. However, based on Frisendal's experience, graph database systems (such as Neo4j and Memgraph, for example) have a strong built-in mechanism for system-level generated UUIDs independent of user-defined attributes. Frisendal explains that there must be recognised two levels of identities and uniqueness in graph databases:

1. Business level keys for identities and concatenation of business keys for uniqueness and
2. The database system automatically generates unique physical-level keys (possibly generated UUIDs or other surrogate keys).

Therefore, Frisendal in [9] proposes the following method for finding and resolving possible hidden functional dependencies as defined in NF6:
Make sure that you have business-level identities on everything – no single attribute concept should be independent of business-level identities, and make sure that you also have business-level uniqueness for every object you create. Check the newly created attributes according to all previous rules.

6 Our approach to the graph database normalisation

Our approach is based on Ambler's three normal forms of object-oriented design, modified in the spirit of Frisendal's proposals. We view data objects as *nodes* in a graph database, and relationships between data objects are *edges* in a graph database.

We worked with the concept of *identity* in the original way; according to our practical experience, data nodes in a graph database (e.g., in the Neo4j system with which we work the most) have a system identity independent of business data values. It is even possible to have two different nodes with the same internal values, but they still be two different nodes. This is because different edges can connect them to other nodes in the database. For this reason, we work with the abstract concept of node identity independently of business data values.

Furthermore, we built one more *level of data types* (or metadata) above the basic *level of data values*. We arranged it symmetrically, following the ideas of Mendeleev's table of chemical elements. Our table has an interesting solution; the first normal form is so specific that its definition directly covers both levels. Of course, Mendeleev's periodic table is also not purely symmetrical but has clear evidence from practice. Our result can be seen in Figure 1.

<table>
<tr><td>data types
(metadata level)</td><td rowspan="2">1st GNF
no repeating</td><td>4th GNF
no sharing datatypes
(inheritance)</td><td>5th GNF
no independent datatypes
(???)</td></tr>
<tr><td>data values
(data level)</td><td>2nd GNF
no sharing data</td><td>3rd GNF
no independent data</td></tr>
</table>

Fig. 1. Graph normal forms.

Now, we proceed with a detailed interpretation of our rules for normalising the graph database. We have also chosen a practical example of *car owners' registration*, which

we will use to demonstrate the benefit of our solution. This example model is only a little simplified compared to reality.

First graph normal form - no repeating data

Rule 1. *A node is in the first graph normal form (1st GNF) when it does not contain a group of repeating values. Groups of repeating values must be extracted into new nodes and linked to the original node by the edges. A database schema is in the 1st GNF when all nodes are in the 1st GNF.*

Definition 1. Let us have a node a, where for $k \geq 1$ (k is the length of the group of repeating data) and $n > 1$ (n is the number of repetitions of the group of repeating data) as $data(a) = \{\dots, x_1^1, \dots, x_1^k, \dots, x_n^1, \dots, x_n^k, \dots\}$, having $\forall i \in (1, \dots, k)$: $type(x_1^i) = type(x_2^i) = \dots = type(x_n^i)$. Then, it is required to extract these repeating data groups from the node a and store them in new nodes b_j for $j \in (1, \dots, n)$ as $data(b_j) = \{x_j^1, \dots, x_j^k\}$ and new edges $[a \rightarrow b_j]$.

Figure 2 presents the situation before normalisation as it can appear printed on paper when collecting software requirements from requesters who want such a database, for example. Figure 3 shows the same model transformed by our rules into the first graph normal form.

Person
+name: String
+surname: String
+birthdate: Date
+country
+city
+street
+1st car label: String
+1st car model: String
+1st car range: Number
+1st car emissions: Number
+1st car made: Date
+1st car producer name: String
+1st car producer fullname: String
+1st car producer country: String
+1st car producer city: String
+1st car producer street: String
+2nd car label
+2nd car model: String
...

Fig. 2. Unstructured database.

In our example, the repeating group of data was data on owned cars, including data on the manufacturers of those cars. In addition, it was heterogeneous data because we are not interested in *emissions* (environmental pollution) with electric cars, unlike internal combustion cars. However, we are very interested in the *range* of electrical batteries (in km or miles). Of course, from the user's point of view, we need to display all data in one place. Nevertheless, if we implement this database according to Figure 2, we will have many problems maintaining and storing data, maintaining data consistency, and making querying very complicated.

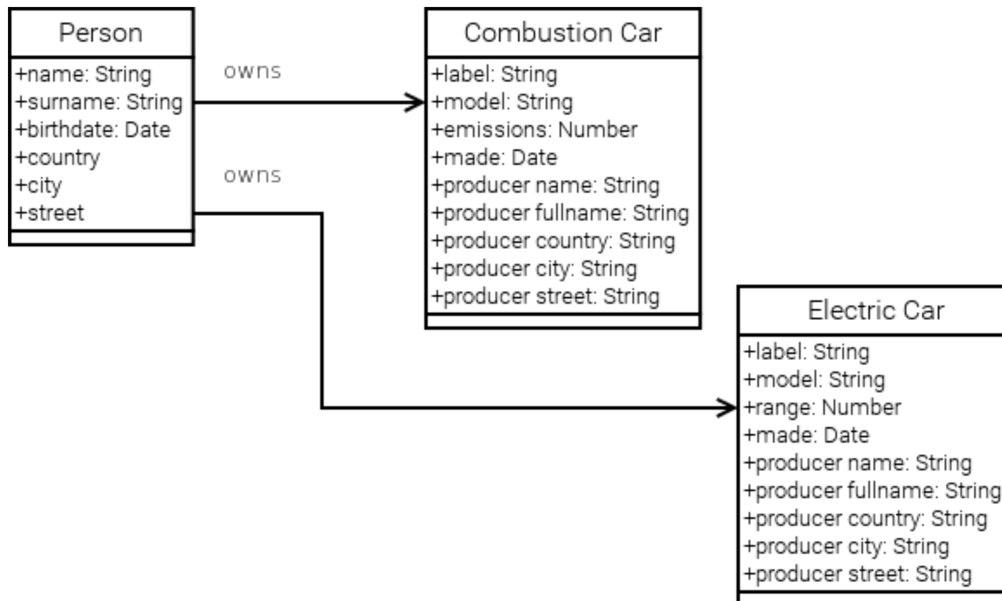

Fig. 3. Database in 1st GNF.

Second graph normal form – no sharing of data

Rule 2. *Nodes are in the second graph normal form (2nd GNF) when they are in the 1st GNF and when they do not share identical data. Shared data must be extracted into new nodes and linked to the original nodes by the edges. A database schema is in the 2nd GNF when all nodes are in the 2nd GNF.*

Definition 2. Let us have two nodes a_1, a_2 for $k \geq 1$ (length of a group of shared data) as $data(a_1) = \{\dots, x_1, \dots, x_k, \dots\}$ and $data(a_2) = \{\dots, y_1, \dots, y_k, \dots\}$ having $\forall i \in (1, \dots, k)$: $x_i \equiv y_i$. Then it is required to extract these shared data from nodes a_1, a_2 to create new node b as $data(b) = \{x_1, \dots, x_k\}$ and create new edges $[a_1 \rightarrow b]$, $[a_2 \rightarrow b]$.

In our example in Figure 4, we have many vehicles (electric and internal combustion) made by the same produced. Therefore, not primarily for memory-saving reasons but rather for greater data safety with the updating, we must ensure that the same data is stored in the database only once in one node and have them linked to other nodes.

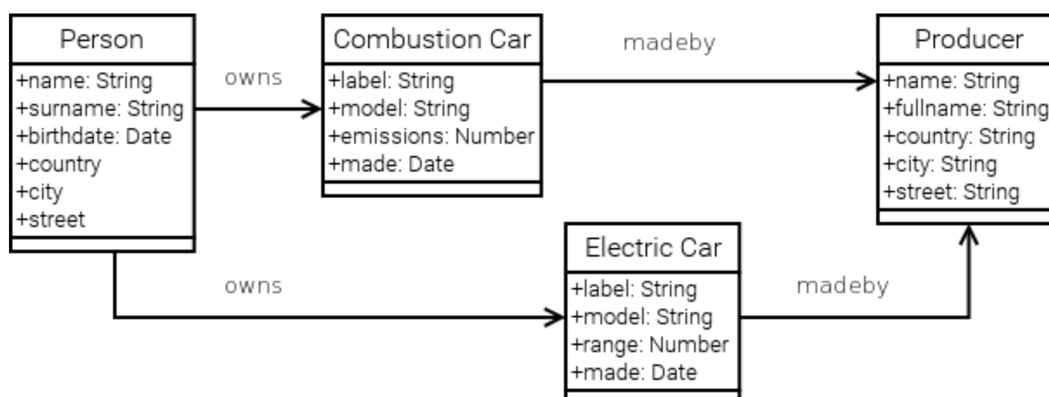

Fig. 4. Database in 2nd GNF.

Third graph normal form – no independent data

Rule 3. *A node is in the third graph normal form (3rd GNF) when it is in the 2nd GNF and when it does not contain a value or a group of values, which have an independent interpretation of the node identity. The independent values must be extracted into a new node and linked to the original node by an edge. A database schema is in the 3rd GNF when all nodes are in the 3rd GNF.*

Definition 3. Let us have a node a for $k \geq 1$ (length of a group of independent data) having $data(a) = \{\dots, x_1, \dots, x_k, \dots\}$, where $\{x_1, \dots, x_k\}$ is a group of independent data. Then, it is required to extract this group of independent data from the node a, and create a new node b as $data(b) = \{x_1, \dots, x_k\}$ and new edge $[a \rightarrow b]$.

In our example in Figure 5, such an independent group is a structured address which we need to use independently, with only the name of the city or country. We can also assume that city names or country names are sometimes changed, although this is rare. However, in any case, the address data values are not controlled and dependent on either the owner or the car manufacturer; they can only place themselves at those addresses.

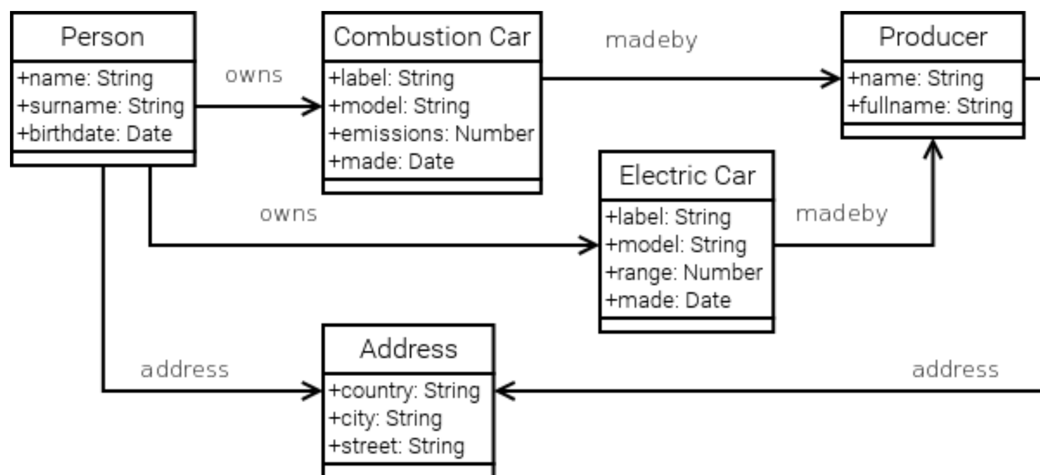

Fig. 5. Database in 3rd GNF.

Fourth graph normal form – no sharing data types

Here it is enough to reuse the definition from the *2nd GNF* and moved from the data level to the type level (i.e. metadata level). Let's look at the result:

Rule 4. *Nodes are in the fourth graph normal form (4th GNF) when they are in the 3rd GNF and when their node types do not share identical data types. Shared data types must be extracted into a new node type and linked to the original node type by inheritance. A database schema is in the 4th GNF when all nodes are in the 4th GNF.* (*The term node type is analogous to the term object class in object-oriented programming.)*

Definition 4. Let us have two node types a_1, a_2 for $k \geq 1$ (length of a group of shared types) as $datatypes(a_1) = \{\dots, type(x_1), \dots, type(x_k), \dots\}$ and $datatypes(a_2) = \{\dots, type(y_1), \dots, type(y_k), \dots\}$ having $\forall i \in (1, \dots, k)$: $type(x_i) \equiv type(y_i)$. Then it is required to extract these shared data types from node types a_1, a_2 to create a new node type b as $datatypes(b) = \{type(x_1), \dots, type(x_k)\}$ and create new inheritance links between node types a_1, a_2, b as $[a_1 \succ b]$ and $[a_2 \succ b]$.

Figure 6 shows an example of using inheritance in our database where we have vehicles with an electric or internal combustion engine. However, the question is how to implement this inheritance. It is easy on the client side of the database application because there we have object-oriented programming languages. Nevertheless, it differs on the server side of the database, and we must find a solution. The concept of *node labels*, part of the forthcoming ISO standard [11], has worked for us, as shown in the data creation code fragment in the CYPHER graph query language [19] in Figure 7. Then, we can work with all nodes of the supertype *Car* or only with the subtype *Electric Car*, as shown in Figures 8 and 9.

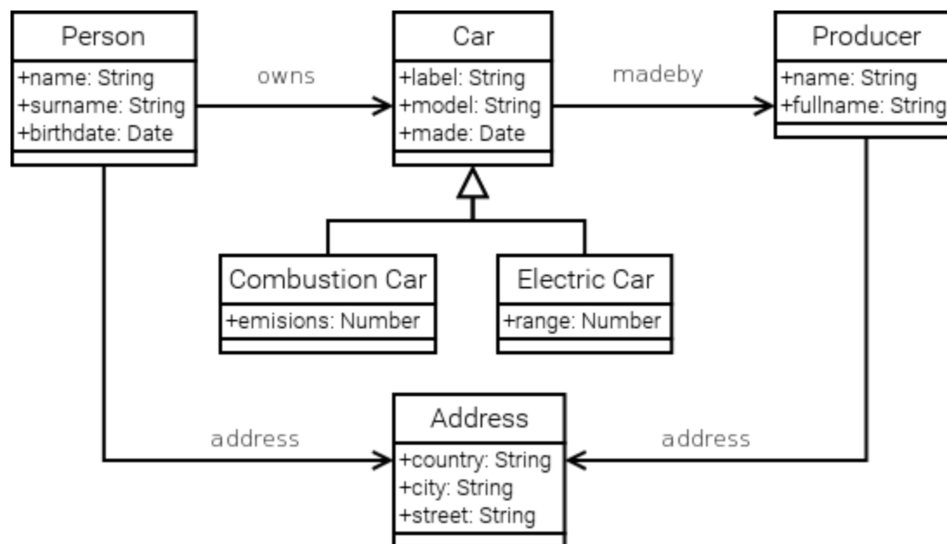

Fig. 6. Database in 4th GNF.

```
CREATE (car1:Car:Combustion {label: "9F7 9987" , model: "Kuga ST-Line"})
CREATE (car2:Car:Electric   {label: "CA 673-998", model: "Tesla S"})
```

Fig. 7. Example of data creation.

```
MATCH (car:Car)-[:madeby]->(prod:Producer)-[:address]->(addr:Address)
RETURN car.label , car.model , prod.name , addr.country
```

Fig. 8. Example of a selection of all kinds of cars with their producers.

```
MATCH (addr:Address)<-[:address]-(owner:Person)-[:owns]->(car:Electric)
RETURN owner.name , owner.surname , addr.city , addr.country , car.model
```

Fig. 9. Example of a selection of only electric cars belonging to any owner.

According to Figure 6, the structure of nodes and edges allows more profitable use of specific graph databases' properties. In addition to the queries known from relational databases, such functions can also be smoothly programmed, which in the case of a relational database would require much greater programming effort, going as far as software add-ons of the business intelligence type. The following two examples on Figures 10 and 11 demonstrate small practical examples supporting our claims.

```
MATCH (addr1:Address)<-[:address]-(owner:Person)-[:owns]->
                                  (car:Car)-[prod:Producer]->(addr2:Address)
WHERE addr1.country = addr2.country
RETURN owner.name , owner.surname , addr1.country , car.model , prod.name
```

Fig. 10. Searching for patriots (i.e. owners of any kind of car manufactured in the same country where they live)

```
MATCH path = ((p:Person)-[*]-(addr:Address))
WITH p , nodes(path) AS nodes
WHERE addr.country = "Japan"
      AND size([node IN nodes WHERE ("Car" IN labels(node))]) <= 1
RETURN (p.name + ' ' + p.surname) as person , nodes as connection_to_Japan
```

person	connection_to_Japan
"Haruto Tanaka"	[(:Person {surname: "Tanaka", name: "Haruto"}) , (:Address {country: "Japan", city: "Tokyo"})]
"John Smith"	[(:Person {surname: "Smith", name: "John"}) , (:Car {model: "RAV4 GR", label: "3A2 5619"}) , (:Producer {name: "Toyota"}) , (:Address {country: "Japan", city: "Toyota City"})]

Fig. 11. Searching for people who have something in common with Japan

The example in Figure 11 searches for everyone somehow related to Japan. Here, the advantages of the graph database are fully shown because the bottom of this query is to find a connection between the nodes defined as "any Person" and "any address in Japan". The result leads through different edges and other internal nodes. We did not have to know the path of edges through internal nodes; instead, we just asked for that unknown path and obtained various kinds of path results:

a) Persons living in Japan have been found (see example in the first row of the table in Figure 11).
b) Another path was also found, as other people living anywhere but owning a car made in Japan (see example in the second row of the table in Figure 11).

To avoid bizarre paths, such as someone owning a car made by the same factory which is also producing yet someone else's car who lives in Japan, we restricted the properties of internal path nodes to at most one car of any car type.

7 Conclusion

This article presented our approach to data normalisation techniques in graph databases. This approach enables the generalisation and explanation of some connections with similar techniques from database design and object-oriented programming. Based on our experience, our design allows development team members to improve the quality of their software engineering work, reduce uncertainty, and improve conceptual consistency to better support graph database properties.

The main theoretical contribution of this article is synthetic to current construction techniques, which provides a solid foundation for future theoretical research and for its practical implementation in some CASE tools that support automated or semi-automated data structure modelling. In detail:

1. We added *inheritance* to the graph normalisation rules, which is currently not supported in graph databases but is present in most programming languages on the client side that access graph databases.
2. We have shown new *conceptual connections* (analogy, metamodelling symmetry) between *data composition* and *data inheritance*.
3. We may have found room for a *new inheritance-like concept* of conceptual modelling (3rd normal form in the metamodel).

It remains to decide what can be hidden under the box 5th GNF in Figure 1. Incomplete classes called *mixins* are offered as possible candidates. Mixins are an object-oriented programming concept that allows a programmer to inject independent code into a class. Mixin programming is a style of software development in which units of functionality are created in a class and then mixed in with other classes. On the conceptual modelling level, this corresponds to the *decorator* (or *wrapper*) design pattern style, where we add more properties to existing object classes. However, since today's graph databases do not even directly support simple inheritance of object classes, we leave this issue open for the future.

Our future research will focus on the further empirical approvement of our claims and programming algorithms for transforming the conceptual model according to our rules in some CASE modelling tools.

Bibliography

1. AMBLER S. W. (2003) Ag*ile Database Techniques: Effective Strategies for the Agile Software Developer*, Wiley Publishing, Inc., ISBN 978-0-471-20283-7.
2. ANGLES R. and GUTIERREZ C. (2008) *Survey of graph database models*, ACM Computing Surveys 40(1), DOI: 10.1145/1322432.1322433

3. BECHBERGER D. and PERRYMAN J. J. (2020) *Graph Databases in Action*, ISBN: 9789332526280
4. CATELL R. and ATWOOD T. (1996) *The Object Database Standard: ODMG-93*, Morgan Kaufmann Series in Data Management Systems, ISBN 978-1558603967
5. CHEN Y. and XING X. (2022) *Constructing Dynamic Knowledge Graph Based on Ontology Modeling and Neo4j Graph Database*, in proceedings of 5th International Conference on Artificial Intelligence and Big Data (ICAIBD) 2022, Chengdu, China, pp. 522-525, DOI: 10.1109/ICAIBD55127.2022.9820199.
6. COAD P. and YOURDON E. (1991) *Object-oriented design*. Yourdon Press and Prentice Hall Inc., ISBN 0-13-630070-7.
7. CODD E. and RUSTIN R. (1972) *Further Normalisation of the Database Relational Model in Database Systems*, Prentice Hall.
8. FALLERI J. R., HUCHARD M. and NEBUT C. (2008) *A Generic Approach for Class Model Normalisation.* In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08). IEEE Computer Society, Washington, DC, USA, 431-434. DOI:10.1109/ASE.2008.66
9. FRISENDAL T. (2022) *Say Hello to Graph Normal Form (GNF)*, Dataversity publication, [https://www.dataversity.net/say-hello-to-graph-normal-form-gnf/]
10. HUANG F. et al., (2023) *KOSA: KO Enhanced Salary Analytics based on Knowledge Graph and LLM Capabilities*, in proceeding of IEEE International Conference on Data Mining Workshops (ICDMW) 2023, pp. 499-505, DOI: 10.1109/ICDMW60847.2023.00071.
11. ISO/IEC FDIS 39075 - Information technology - Database languages (2024) *GQL standard under development*, [https://www.iso.org/standard/76120.html]
12. ISO/IEC 19505 - Information technology (2017) *OMG UML*, [https://www.iso.org/standard/52854.html]
13. ISO/IEC 24707 - Information technology (2023) *Common Logic - A framework for a family of logic-based languages*, [https://www.iso.org/standard/66249.html]
14. LISSANDRINI M., MOTTIN D., PALPANAS T., and VELEGRAKIS Y. (2020) *Graph-Query Suggestions for Knowledge Graph Exploration*, in Proceedings of The Web Conference 2020 - WWW '20, DOI: 10.1145/3366423.3380005
15. LO S. H., SHIUE Y. C. and LIU K. F. (2018). *Seven steps for object-oriented normalisation in class diagrams: Example of jigsaw puzzle concept for image retrieval*. Journal of Applied Science and Engineering. 21. 463-474. DOI:10.6180/jase.201809_21(3).0018.
16. MEIER A. and KAUFMAN M. (2019) *SQL & NoSQL Databases*, Springer, ISBN 978-3-658-24548-1
17. MOLHANEC M. (2019) *Conceptual Normalisation in Software Engineering*, in: Proceedings of EOMAS 2019, LNBIP 366, Springer International Publishing, pp. 18–28, DOI: 10.1007/978-3-030-35646-0_2
18. NAIBURG E. J. and MAKSIMCHUK R. A. (2003) *UML for Database Design, Chapter 7 - Database Design Models - the UML Profile for Database Design*, Addison Wesley Longman, Inc., ISBN 0201721635.
19. ROBINSON I., WEBBER J., and EIFREM E. E. (2015) *Graph Databases - New Opportunities for Connected Data*, O'Reilly Media, Inc., ISBN 978-1-491-93200-1
20. ZHOU B., LI X., LIU T., XU K., LIU W., BAO J. (2024) *CausalKGPT: Industrial structure causal knowledge-enhanced large language model for cause analysis of quality problems in aerospace product manufacturing*. Advanced Engineering Informatics vol. 59 2024, DOI: 10.1016/j.aei.2023.102333.