POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

From OntoUML model to graph database - completion of the workflow, case study and benchmark

Politecnico di Torino

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Supervisors
Dr. Michal Valenta
Prof. Luca Ardito
Contributor
Ing. Jiří Zikán

Candidate
El Mahdi Affaoui

June 2025

Abstract

Software development faces increasing complexity as systems grow in scale and sophistication, requiring developers to manage layered architectures and diverse technologies. Model-Driven Development (MDD) addresses these challenges by elevating the design process through high-level abstractions. This thesis builds upon the work of Ing. Jiří Zikán, who developed a tool to automatically generate Neo4j graph database triggers from OntoUML models. Expanding on his research, this work evaluates the tool's real-world applicability and performance through a practical case study involving Uniqway, a car-sharing application. The thesis outlines improvements made to the transformation pipeline, including support for models imported from Visual Paradigm. Performance benchmarks assess the effectiveness and efficiency of the generated triggers, comparing system behavior with and without their application. The results offer insights into the correctness, scalability, and operational impact of integrating OntoUML-based constraints into graph-based systems, thereby contributing to the broader goal of enhancing software reliability through model-driven approaches.

Acknowledgments

FIRST AND FOREMOST, I am deeply grateful to my supervisor, Dr. Michal Valenta, for his invaluable guidance, encouragement, and patience. His expertise and insightful feedback have been fundamental in shaping this work. Coming into this thesis completely inexperienced, being my one first ever, I was fortunate to benefit from his steady mentorship and clarity throughout the process. I would also like to extend my sincere thanks to Ing. Jiří Zikán, whose own master's thesis laid the foundation to mine and whose suggestions helped me start my work.

I'm truly grateful to all the new people I met during my time in Prague, some of whom ended up becoming a real part of my everyday life. The time we spent together, playing games, exploring the city, chasing sunsets or simply getting through the ups and downs of life, made this experience so much richer and more enjoyable. Since the first they I loved the city but spending a lot of time on my own, the city appeared cold and my love for it felt unrequited. But meeting you changed that. You brought warmth, laughter, and a sense of belonging that slowly turned this place into something that truly felt like home.

Of course, heartfelt thanks go to my family back home, especially my mother, whose constant thought and care were a source of comfort across the distance. Her support helped me avoid homesickness and reminded me that no matter how far I was, I was never alone. I'm equally thankful to all my friends who stayed in touch despite the distance.

Coming from an immigrant community where many face economic hardships, pursuing higher education is a significant achievement, not just for me, but for many others who, despite the challenges, are striving for a better future. This thesis represents not only my personal effort but also the collective determination of those who, like me, are working hard to create new opportunities and break cycles. It reminds me that I am part of a larger story, one of resilience and hope.

Finally, I would like to thank the city of Prague itself. Here, for the first time, I experienced what it truly means to change one's life: to start anew, to live independently, and to grow. It was here that for the first time I marched to raise awareness about the necessity of a free Palestine for all, *from the river to the sea*. It was here that I built the confidence and strength I'll carry forward, ready to face future challenges and take ownership of my life and the decisions ahead.

Ringraziamenti

PER PRIMA COSA, desidero esprimere la mia più profonda gratitudine al mio relatore, il Dr. Michal Valenta, per la sua guida preziosa, il suo incoraggiamento e la sua pazienza. La sua competenza e i suoi suggerimenti puntuali sono stati fondamentali per plasmare questo lavoro. Essendo alla mia prima tesi e del tutto inesperto, ho avuto la fortuna di poter contare sulla sua costante disponibilità e chiarezza. Desidero inoltre ringraziare sinceramente l'ingegner Jiří Zikán, la cui tesi magistrale ha rappresentato la base su cui si fonda il mio lavoro, e i cui consigli iniziali mi hanno aiutato a partire con il piede giusto.

Sono profondamente grato a tutte le nuove persone conosciute durante il mio periodo a Praga, alcune delle quali sono diventate parte essenziale della mia quotidianità. Il tempo trascorso insieme, tra giochi, esplorazioni, tramonti e confidenze, ha reso questa esperienza molto più ricca e significativa. Fin dal primo giorno ho amato questa città, ma la solitudine la faceva spesso sembrare fredda. Incontrare voi ha cambiato tutto: avete portato calore, risate e un senso di appartenenza che l'ha trasformata in qualcosa che finalmente potevo chiamare casa.

Un ringraziamento speciale va, naturalmente, alla mia famiglia, in particolare a mia madre, la cui premura costante è stata un conforto prezioso nonostante la distanza. Il suo sostegno mi ha aiutato a non cedere alla nostalgia, ricordandomi che, per quanto lontano, non ero mai davvero solo. Sono ugualmente riconoscente agli amici con cui ho mantenuto i contatti nonostante la distanza.

Venendo da una comunità di immigrati dove molti affrontano difficoltà economiche, proseguire gli studi universitari è un traguardo importante, non solo per me, ma per tutti coloro che, nonostante le sfide, si impegnano per un futuro migliore. Questa tesi rappresenta non solo il mio sforzo, ma anche la determinazione collettiva di chi lavora per creare nuove opportunità e spezzare i cicli. Mi ricorda che faccio parte di una storia più grande, fatta di resilienza e speranza.

Infine, vorrei ringraziare la città di Praga stessa. È qui che, per la prima volta, ho scoperto cosa significa cambiare vita: iniziare da capo, vivere da solo, crescere. È qui che ho manifestato per la prima volta per una Palestina libera per tutti, *dal fiume al mare*. Ed è qui che ho costruito la fiducia e la forza che porterò con me, pronto ad affrontare le sfide future e a prendere in mano la mia vita.

Contents

List of Figures

List of Tables

List of source codes

Introduction

Motivation

Software development is an inherently complex and evolving discipline, driven by the need to build reliable, scalable, and maintainable systems in an increasingly fast-paced environment. As applications grow in size and sophistication, developers must manage intricate architectures, multiple layers of abstraction, and a wide range of technologies. This complexity often leads to increased development time, higher costs, and greater potential for errors. To address these challenges, the industry continually seeks methodologies and tools that can improve productivity, reduce human error, and enhance the overall quality of software. One such approach is Model-Driven Development (MDD), which aims to shift focus from low-level implementation details to high-level design and abstraction.

Objectives of the Thesis

This thesis builds upon the work of Ing. Jiří Zikán, whose master's thesis focused on the development of a tool for automatically generating triggers for the Neo4j graph database management system based on OntoUML models [Zik23]. His research introduced a novel approach for bridging conceptual modeling and runtime enforcement of constraints in graph-based systems, laying the groundwork for practical applications of OntoUML in NoSQL environments. The present work should be regarded as a continuation and extension of that research. In particular, this thesis aims to evaluate the real-world applicability of the trigger generator through a case study that demonstrates its integration in a functional system. It further assesses the tool's quality and performance by conducting benchmarks that compare system behavior with and without the generated triggers, providing insights into both the correctness and efficiency of the approach.

Related Works

In addition to Jiří Zikán's previously cited work on graph databases derived from OntoUML, several other relevant contributions in this area include the studies by Pokorný, Valena, and Zikán [PRV+24, PVZ25, PVK17]. Further related research includes work by Rybola et al. on transforming OntoUML models into relational databases [RP17], as well as Grievink's thesis on generating Java code from OntoUML models [Gri24].

Contents

This thesis is structured into six main chapters and eight appendices, each addressing a key aspect of the research.

Section 1: Background introduces the foundational concepts necessary to understand the rest of the work. It begins with an overview of OntoUML, including its conceptual basis and its integration with the UFO (Unified Foundational Ontology)

framework. The chapter then shifts focus to graph databases, with a particular emphasis on Neo4j, the Cypher query language, support for triggers through the APOC library, and the practical advantages and use cases of graph-based storage.

Section 2: Methodology presents the transformation process from OntoUML models to a Neo4j graph database. It discusses the motivation behind this transformation, reviews the methodology adopted. This includes the development of a significant update to the transformation pipeline to support models imported from Visual Paradigm.

Section 3: Case Study applies the proposed methodology to a real-world scenario involving Uniqway, a car-sharing application. The chapter provides an overview of Uniqway, explains the data selection and preparation steps, and describes how this data was integrated into a Neo4j graph database.

Section 4: Benchmark Design and Implementation details the design of performance benchmarks used to evaluate the effectiveness and efficiency of the proposed transformation and data model.

Section 5: Evaluation and Results presents the results obtained from the benchmarks. It outlines the evaluation methodology, discusses the key findings, and provides an interpretation of the results in relation to the goals of the thesis.

Section 6: Conclusion summarizes the main contributions of the work, discusses its limitations and challenges encountered, and proposes directions for future research.

The **Appendices** provide supplementary materials. **Appendix A** lists the comments generated by the tranformer, **Appendix B** provides the sizes of the tables used, **Appendix C** provides the cardinalities fro each of the main labels in the different graph database instances, **Appendix D** includes the Cypher script used to populate the Neo4j database, **Appendix F** provides the specifications for the machine used during the tests, **Appendix G** provides the C# code for the JSON importer developed to bridge the gap from Visual Paradigm to the constraint generator, **Appendix H** contains images of the full execution plans of the benchmark queries, **Appendix E** shows a summary of the benchmark tests used.

SECTION 1

Background: OntoUML and Graph Databases

1.1 OntoUML

OntoUML is a structural modeling language developed initially by Guizzardi in his Ph.D. thesis [Gui05]. It is grounded in the Unified Modeling Language (UML), which it extends by introducing ontologically well-founded types and constraints based on the Unified Foundational Ontology (UFO) [GG+22]. OntoUML enables the construction of semantically rich models by ensuring that model elements correspond to well-defined ontological categories.

The Unified Foundational Ontology (UFO), on which OntoUML is based, is a comprehensive axiomatic system that draws upon insights from analytic philosophy and cognitive science to support conceptual modeling and domain analysis [GSAG21]. It provides a formal distinction between different types of entities, such as objects, events, properties and relations, and defines meta-properties like identity, rigidity and dependence, which OntoUML leverages to classify modeling constructs [GSAG21].

The consistency and expressivity derived from its ontological grounding, allows OntoUML to offer benefits in tasks such as domain understanding, communication among stakeholders [ea23], and system design validation [ea10].

1.1.1 Overview of OntoUML

After its introduction in the aforementioned Ph.D. thesis from 2005 by Guizzardi [Gui05], OntoUML was continually researched and expanded in successive papers [G+18] [GSAG21] [FSGA19] leading to the definition of its newest incarnation called OntoUML 2.0 and to the expansion of its practical applications across various domains.

Among the use cases that have been researched for OntoUML we can cite Braga et al.'s paper on transforming the OntoUML model to the logic-based language Alloy to perform model validation [ea10]. Of course, it is also being used to generate databases, both relational [RP17] and graph-based, as in the research-line this thesis is based and expands upon [Zik23] [PVK17] [PRV+24] [PVZ25]. While regarding the use of OntoUML as a reference ontology for the development of application Grievnik's master thesis researched a transformer from OntoUML to Java code to ensure the semantics of the model are respected during development.[Gri24]

1.1.2 OntoUML's integration of the UFO framework

OntoUML's metamodel is deeply integrated with the Unified Foundational Ontology (UFO), providing a robust framework for classifying entities based on ontological distinctions. A full taxonomy of the UFO is shown in **Figure 1** [GG+22].

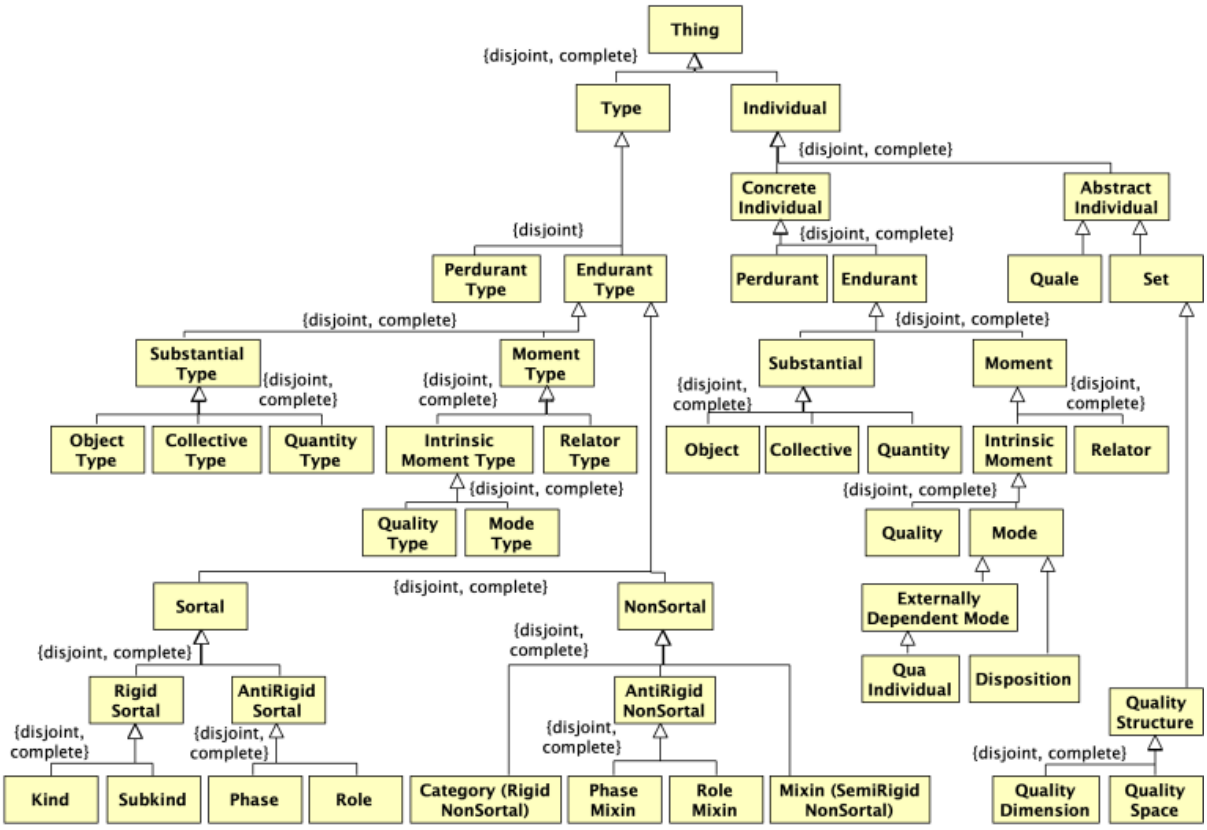

Figure 1: UFO taxonomy of endurant types adapted from Guizzardi (2022).

Each type is characterized by meta-properties derived from UFO, such as rigidity, dependence, and identity. This structured approach ensures that conceptual models are semantically precise and align with real-world ontological distinctions [GSAG21].

UFO divides types of entities in two fundamental distinctions, **endurants** and **perdurants**. Endurants are individuals that exist continuously in time and can accumulate changes through their existence. On the other hand, perdurants represent events at a determinate time, as such they exist only in the past and cannot really change[GG$^+$22].

A rich taxonomy of endurants was introduced by Guizzardi et al. in 2021 [GSAG21] based on several meta-properties. The first meta-property by which we classify them into their taxonomy is **Substantiality**, a substantial entity is an entity that has independent existence, as opposed to a moment which is an entity whose existence is dependent on another[GSAG21].

For substantial types the categorization makes use of the concepts of **Sortality** and **Rigidity**. A **sortal** type is a type that defines the properties that define the identity of its instances (properties that can't be shared by different objects), while a non-sortal type aggregates properties shared between instances of different sortals [GSAG21]. A **rigid** type is a type that necessarily applies to its instances, an anti-rigid type classifies only contingently to its instances and exists only as a subtype of a rigid type; an additional category of semi-rigidity exists for types that generalize both rigid and anti-rigid subtypes. In other words we can say that a type is rigid if its instances can only lose its properties by ceasing to exist[GSAG21]. Depending on these properties types can be divided in different categories, called **sterotypes**, or metetypes. The most important stereotypes are the following:

- *Kind*: a substantial, sortal and rigid type
- *Subkind*: similar to a *Kind* but it inherits from a *Kind* or another *Subkind* while *Kind* is used only for the most general type
- *Phase* and *Role*: both substantial, sortal and antirigid types. The difference between them stands in how Phase depenends on intrinsic conditions while role's classification condition is relational
- *Category*: substantial, non-sortal and rigid type that defines some essential properties of its instances without providing any identity for them
- *Phase Mixin* and *Role Mixin*: non-sortal and anti-rigid substantial types that generalize and provide contingent properties respectively of Phases and Roles of different kinds
- *Mixin*: non-sortal and non-rigid type that generalizes properties that are essential for some instances and contingent for others
- *Quality*: a moment type that represents a property of the substantial object on which it depends
- *Relator*: a moment type which ties together the multiple substantial entities on which it depends

[GG+22]

1.2 Graph Databases

Graph databases are a class of NoSQL databases designed to represent and store data using graphs as the underlying data structure with elements represented as vertices and edges representing the relationships between them. Unlike relational databases, which model data in tabular formats, graph databases focus on the explicit representation of relationships between entities. Where relational databases would require multiple slow recursive joins to follow paths composed of subsequent relationships, graph databases can handle relationships natively and traverse the graph with very fast speeds and efficient memory usage thanks to theirs native graph processing engines that avoid the need for costly joins [AG08, RWE15].

Graph databases are particularly well-suited for domains where relationships carry as much meaning as the data itself, and where traversing complex connections is common such as social networks, spatially embedded networks (for example for transportation), recommendation systems and semantic data. [RWE15].

Over the years, various graph data models and query languages have emerged. Among them, the *labeled-property graph model* and the *RDF triple model* are most prominent. Property graphs allow multiple labels per node and arbitrary properties on both nodes and relationships, offering a flexible schema-less structure [RWE15].

Graph databases can be divided into *native* and *non-native* stores. Native graph databases use purpose-built storage and indexing mechanisms optimized for graph processing, while non-native systems provide graph capabilities over general-purpose stores. Native systems like Neo4j offer index-free adjacency, which enables constant-time traversals of relationships between nodes, a significant performance advantage for complex queries [RWE15].

1.2.1 Introduction to Neo4j

Neo4j is one of the most mature and widely used native graph databases. Initially released in 2007, it implements the *labeled-property graph model* and supports the declarative query language *Cypher*, which is tailored for expressing graph patterns in an intuitive way [RWE15].

In addition to its expressive query capabilities, Neo4j includes limited support for data integrity and efficient retrieval through uniqueness constraints on node properties and indexes. Neo4j also supports composite and full-text indexes, enabling fast lookup of nodes or relationships based on one or more properties, which is crucial for performance in large-scale graph queries.

Neo4j ensures transactional consistency (ACID) and allows further extensibility through the *APOC* (Awesome Procedures on Cypher) library. It also provides a rich ecosystem for data visualization and integration with other technologies, making it a preferred platform for applications involving graph-based reasoning, recommendation systems, or semantic analysis [Neo].

1.2.2 Cypher Query Language

Neo4j uses the Cypher language for its queries. It was developed specifically for use on Neo4j and it makes use of a declarative approach similar to SQL, where the query focuses on the data particulars of the request while the execution plan, which forms the algorithm through which the query will be executed, will be generated in a subsequent interpretation by the engine. Also from SQL it borrows some of the query syntax which it simplifies and expands for ease of use in the new context. One of the most relevant new feature is its arrow notations for relationships, in the form `(a)-[:R]->(b)` to denote a relation of type R between a and b. Cypher also is adapted to Neo4j by not requiring to follow any schema, allowing for example to add or remove properties to specific nodes without any consideration for other nodes in the graph[Cyp]. Unless, of course, we enforced a certain schema through indices and constraints like the method we demonstrate here.

Following is an example to compare the syntax of SQL and Cypher:

```sql
SELECT actors.name
FROM actors
LEFT JOIN acted_in ON
    acted_in.actor_id =
    actors.id
LEFT JOIN movies ON
    movies.id =
    acted_in.movie_id
WHERE movies.title = "The
    Matrix"
```

```
MATCH (actor:Actor)-[:ACTED_IN]->
    (movie:Movie {title: "The
        Matrix"})
RETURN actor.name
```

1.2.3 Transaction life cycle

In Neo4j, all operations that access the graph, indexes, or schema are encapsulated within transactions to uphold the ACID properties — Atomicity, Consistency, Isolation, and Durability. This ensures that each unit of work is executed reliably: either fully completed or not applied at all, leaving the database in a consistent state. Transactions in Neo4j are single-threaded and confined, meaning they operate independently without sharing state with other transactions. This design avoids complex concurrency issues and simplifies reasoning about transactional behavior.

The life-cycle of a transaction in Neo4j is straightforward: it begins, executes a series of read or write operations, and then finishes by either committing or rolling back. A commit applies all the changes to the database, making them durable and visible to subsequent transactions, while a rollback undoes all changes made during the transaction. Despite each transaction running in isolation, Neo4j supports concurrent transactions, even for write operations. To handle potential race conditions and ensure consistency, Neo4j uses a locking mechanism: write locks are acquired on nodes, relationships, or properties being modified. If two transactions attempt to write to the same data, one will wait or fail, depending on lock availability and timing. This mechanism guarantees *serializability*, the highest isolation level, ensuring that concurrent transactions yield the same result as if executed sequentially. Nevertheless the default isolation level is *read-commited isolation*, in which a locks are not acquired for reading, allowing another transaction to write to an element before a transaction that reads that same element has finished. This is a weaker level of isolation but is sufficient for most use cases and provides greater performance. [Neo]

1.2.4 Optimization in Neo4j

As previously stated, a Neo4j query is initially written as a Cypher query in the form of a string. This query represents the pattern to be matched, along with statements describing operations to be performed on the database. The query is parsed into an internal structure, which is then passed to the query optimizer. The optimizer analyzes the structure by comparing it with statistics about the contents of the database and existing indexes. It then assigns logical operators to the various components of the query, aiming to identify the most efficient execution strategy in the current context. This logical plan is finally converted into a physical plan, which is the actual executable form of the query. This physical plan is executed by the Cypher runtime.

Various options for query execution can be specified by prepending the statement `CYPHER query-option [further-query-options]` to the query. For example, writing `planner=dp` instead of `planner=idp` allows the use of the dynamic programming (DP) planner rather than the IDP (Interleaved DP) planner. The DP planner explores a larger search space of possible execution plans, potentially identifying more efficient plans at the cost of increased planning time. This can be advantageous when a query is executed frequently, such that the time saved during execution offsets the longer planning time.

The Cypher query planner relies on accurate statistics to create efficient execution plans. These statistics are automatically maintained and updated by Neo4j as the database evolves. Examples include the number of nodes per label, relationships per type and per pair of node labels, and index selectivity.

Cypher supports three query runtimes: **slotted**, **pipelined**, and **parallel**. The `slotted`

runtime, which is the default in the Neo4j Community Edition, is optimized for performance by using fixed memory slots for each variable. The `pipelined` runtime allows intermediate results to be passed between operators in a streaming fashion, which reduces memory usage and can improve performance for certain queries. The `parallel` runtime extends the pipelined approach by executing parts of the query plan in parallel threads when possible, further improving performance on multi-core systems. The runtime can be selected using a query option such as `runtime=slotted`.

To inspect and debug query performance, Cypher provides two keywords: `EXPLAIN` and `PROFILE`. The `EXPLAIN` keyword parses and plans the query, returning the execution plan without executing the query itself. This is useful for understanding how Neo4j intends to execute the query. The `PROFILE` keyword both executes the query and returns the detailed execution plan, including runtime statistics such as the number of database hits and rows processed at each step.

When executing a query with the `EXPLAIN` or `PROFILE` on the Neo4j browser client the execution plan is returned in an image format as shown in **Figure 2**. Each operator is shown as a rectangular block on the header of which the name of such operator can be read. The body of the block is divided in two parts. On top we have the schema of the row: the elements the operator receives as variable and on which can operate. The bottom part represents the actual operations being performed on the elements of the row. Additionally on the connections between blocks we can see a numbered representing how many rows the query planner expect to receive based on the statistics, it is according to this number that the optimizations are performed.

The execution plan does not need to be recalculated every time Neo4j executes a query. Neo4j maintains a cache of the last 1000 execution plans to avoid redundant planning. This cache operates at the level of the Abstract Syntax Tree (AST) resulting from query parsing. Consequently, even if a query is written in a syntactically different but semantically equivalent form, or if it contains different literal values, the cached plan can still be reused. A divergence threshold, set to 75% by default, limits how much the underlying database statistics are allowed to change before the cached execution plan is considered invalid and a new one must be generated.

1.2.5 Apoc support for triggers

The APOC (Awesome Procedures on Cypher) library is a powerful and widely-used extension for Neo4j that provides a rich set of procedures and functions to enhance Cypher's capabilities. One of its key features is support for triggers, which enable users to define custom logic that automatically executes in response to write operations—such as the creation, update, or deletion of nodes and relationships. These triggers allow for reactive behavior within the database, supporting use cases like data validation, automatic property management, auditing, and the enforcement of business rules. Defined and managed using Cypher, APOC triggers offer a flexible mechanism for extending Neo4j's native functionality [Apo]. This library is leveraged by the constraint generator developed by Jiří Zikán, which uses APOC to automatically derive and apply constraints to a Neo4j graph based on an OntoUML model [Zik23].

The function use for trigger instantiation is `apoc.trigger.add`. It has the following syntax `apoc.trigger.add(name, statement, selector [, config ])` where the argument `name` is a string representing the name chosen for the trigger being created, `statement` is a string variable representing the query being execute

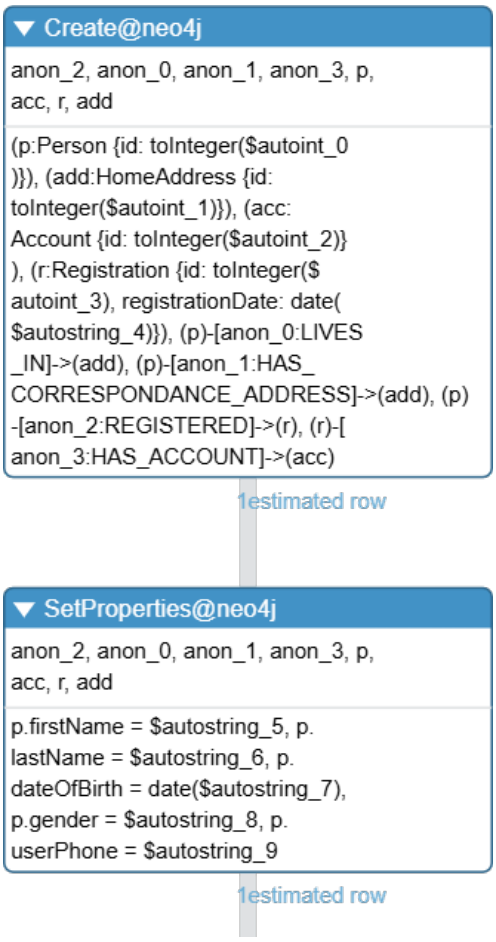

Figure 2: Example of portion of execution plan

by the trigger and `selector` is a `map` of the type phase : PHASE through which the phase of the transaction in which the trigger is activated, e.g. phase: 'before' makes the trigger execute before the commit[Apo].

In addition `apoc.trigger` provides additional data structures and functions to make the triggers more efficient for common uses. The data structures can be consumed as parameters by the statement (e.g. $transactionId returns the id of the triggering transaction and allows us to use it in the query) and include most importantly collections of modified nodes, relationships and property, e.g. $createdNodes is the set of all nodes created in the transaction. A common way to employ these parameters is by making use of the statement UNWIND (i.e. UNWIND ($createdNodes) as createdNode) makes the rest of query execute for each created node. A useful faction is `apoc.trigger.nodesByLabel(labelEntries, label)` where `labelEntries` is a collection fo nodes, for example those returned by parameter such us `$createdNodes`, and `label` is string representing the label we want to filter from the collection. This allows us to easily and efficiently make triggers that operate only nodes with a specific label.

1.2.6 Use Cases and Advantages of Graph Databases

Graph databases offer significant performance advantages over traditional relational databases, primarily due to their use of index-free adjacency. In graph databases, each node directly references its adjacent nodes, eliminating the need for expensive join operations. In contrast, relational databases rely on joins whose performance degrades as table sizes grow. This fundamental difference gives graph databases superior scalability, especially as data complexity increases. Additionally, systems like Neo4j provide high flexibility, making them more adaptable to evolving application requirements. Unlike relational databases, which often require time-consuming and costly schema migrations, graph databases can accommodate structural changes with minimal overhead.[RWE15]

Thanks to these characteristics, graph databases are particularly well-suited for domains where data is naturally represented as a network of relationships. A prominent example is social networks, where users and their interactions form a complex, highly connected graph. Graph databases can efficiently extract insights such as mutual connections, communities, or influence chains. In geospatial applications, graphs can model real-world adjacency (e.g., roads, intersections, or regions), making it possible to implement fast and dynamic routing algorithms or analyze spatial proximity. Another important use case is network and IT infrastructure management, where devices and their connections can be monitored and optimized as part of a live, updatable graph. In all these cases, the ability to traverse relationships quickly and intuitively is key—and graph databases excel precisely in that area. [RWE15]

SECTION 2

Methodology: From the OntoUML Model to a Neo4j Database

This thesis makes use of the constraint generator developed by Jiří Zikán in his master's thesis [Zik23] and expands the workflow by integrating it with the UML modeling Visual Paradigm.

2.1 Motivation for Transformation

Having such a generator allows for fast and reliable model driven development. As cited in the original thesis, model driven development is still not as common in great part because of the lack of tools like this that could help better define the workflow by automating part of the work necessary in actually implementing the model previously developed.

2.2 Transformation Approach

The theory of the generator involves first defining a mapping from OntoUML constructs to the Neo4j constructs that will be used to manage them and ensure that the model is respected. The mapping derived is shown in **Table 1**.

OntoUML constructs	Neo4j constructs
Classes	Labels
Attributes	Integrity Constraints on the properties of vertices
Stereotypes	Integrity Constraints on the combination of labels
Generalizations	Integrity Constraints on the combination of labels
Associations	Integrity Constraints on the presence of edges

Table 1: Mapping between OntoUML and Neo4j constructs from Jiří Zikán's master's thesis

Following this mapping the generator was developed to create all the required integrity constraints for each instance in which these constructs are applied. The mapping shows already why this approach is very promising since all the constraints required involve computations which Neo4j is very efficient in doing, thanks to it's native handling of relationships and multi-label nodes which allow fast checking of constraints on types.

The transformer generates the following constraints:

- **PROPERTY_MUST_BE_OF_DATATYPE**: checks that the values of the property of vertices respect the types defined in the model for the class to which the vertex belongs to
- **PROPERTY_MUST_BE_PRESENT**: checks that the properties specified as mandatory for a certain class are indeed present in vertices that bear the corresponding label
- **PROPERTY_MUST_BE_UNIQUE**: ensures uniqueness of identity attributes of a class among vertices with the corresponding label
- **LABEL_MUST_BE_IN_COMBINATION**: this constraint is extracted both from analysis on stereotypes and on generalizations. In the case of the stereotypes, it ensures that a label for a NonSortal class can be applied to a vertex only if it is in combination with at least one label for a sortal class that is intransitively related to it. For the purpose or this constraint a related sortal is defined as a sortal that is a subclass of the NonSortal or of one of it's superclasses, and an intransitively related sortal is a related sortal that is not a subclass of another related sortal. While for the generalizations it ensures that an entity belonging to a class also belongs to all of its superclasses and that the superclass of a covering generalization set exist always with at least on of its subclasses.
- **LABEL_CANNOT_BE_IN_COMBINATION**: another constraint extracted both from analysis on stereotypes and on generalizations. On the stereotypes it ensures that a vertex does not belong to more than one identity providing class while on generalizations it ensure that a vertex cannot have multiple labels belonging to disjoint generalization set.
- **RELATIONSHIP_MUST_BE_PRESENT**: ensures that if class has a lower bound for a relationship greater than 0 this relationship is always present.
- **RELATIONSHIP_MUST_BE_LIMITED**: ensures that the higher bound for the cardinality of classes in the relationships are respected.

After the transformatons the program generates to files: one called `comment.txt` contains a list of the constraints by name as shown in the previous list and provides an accessible way for the user to check what are the constraints that resulted from the OntoUML model; another file called `triggers.txt` contains the Cypher script that can be used on the Neo4j server to instantiates the triggers.

In the `triggers.txt` file, each constraint is translated into an APOC trigger, with the exception of the *PROPERTY_MUST_BE_UNIQUE* constraint. This specific constraint can be directly enforced using Neo4j's built-in constraint mechanisms, which allow the declaration of a uniqueness constraint on a property of a node or relationship.

Uniqueness constraints in Neo4j are implemented internally by creating a backed unique index on the specified property. This ensures that no two nodes (or relationships, in newer versions) with the given label can have the same value for that property. If a user attempts to insert or update a node in a way that violates this constraint, Neo4j will reject the operation and raise an error. This enforcement happens at the database engine level, making it more efficient and reliable than application-level checks or triggers.

The template used to generate this type of constraint is provided in **Code 1**. For all other constraint types, which cannot be enforced using native mechanisms, equivalent APOC triggers are generated. Details on how these triggers are defined and implemented can be found in **Section 1.2.5**, while concrete examples from the use case are shown in **Section 4.2**.

```
1 //CONSTRAINT: PROPERTY_MUST_BE_UNIQUE
2 //INPUT: labelName, propertyName
3 CREATE CONSTRAINT {labelName}_{propertyName}_must_be_unique IF NOT
   ↪ EXISTS
4 FOR (node:{labelName}) REQUIRE node.{propertyName} IS UNIQUE
```

Listing 1: Script for constraint PROPERTY_MUST_BE_UNIQUE

2.3 Updating the Transformer with Importer from Visual Paradigm

The program makes use of an internal data structure created through C# code to represent the OntoUML model to be transformed. To further streamline the workflow I here expanded the original transformer by adding to it an importer that allows to create the original model in a visual modeling program and import it into the transformer.

The modeling program chosen is Visual Paradigm, a software platform used for modeling, designing, and managing software development projects. It provides tools for creating diagrams like UML, BPMN, ERD, and more, helping teams visualize system architecture, business processes, and data structures. The key reason for chosing this program is that a plug in has been developed for it that allows efficient use of OntoUML constructs.[Ontb] A related work is the OntoUML JSON2Graph decoder which also decodes the json file exported from Visual Paradigm but then transforms it into ONTOUML Vocabulary, an implementation of OntoUML in the Web Ontology Language [Bar] [Onta].

In this case I designed a JSONImporter class to parse and translate the JSON file exported from Visual Paradigm into the internal representation used for further processing within the Neo4jConstraintGenerator system. Its primary responsibility is to read a JSON file, analyze its contents, and reconstruct the OntoUML model as a collection of well-defined C# objects, including classes, associations, generalizations, and generalization sets. This enables a seamless transition from a textual data exchange format to a typed, object-oriented model that can be used for logic validation or conversion to database constraints.

Upon instantiation, the JSONImporter is provided with a file path pointing to the target JSON file. The CreateModelFromJSON() method orchestrates the entire import process. It begins by reading and parsing the JSON document, and accesses `model.contents` which contains all the information about the elements of the models, included data about the graphical dispositions of the elements in the Visual Paradigm diagram which can be safely ignored for the purposes of this project. The program separates the relevant elements into categories based on their OntoUML types (i.e., Class, Generalization, Relation, GeneralizationSet, DataType). Special attention is given to distinguishing between regular classes and data types, which are represented in the

JSON object as Classes with the stereotype `datatype` but must be stored and processed differently due to their semantic roles in the model.

The categorization process is of fundamental importance since the JSON models uses references through ids. Categorizing the elements into lists allows us to process them in order so that every time a reference by id is being interpreted, the relative element has already been processed and can be accessed via id through a dictionary. Since classes depend on datatypes for the types of their properties, generalizations and relations refer to classes and generalization sets depend on generalizations one of the possible orderings and the one is in the present code is the following: datatypes, classes, generalizations, relations, generalization sets.

The importer uses dedicated lookup functions to interpret OntoUML-specific semantics. For example, lookupStereotype() maps string representations of OntoUML stereotypes (e.g., kind, role, relator) to corresponding enum values used internally. Similarly, lookupMultiplicity() translates multiplicity constraints from their string format (e.g., ”1”, ”0..*”) into structured Multiplicity objects. Class attributes, including their names, types, and identity markers, are also extracted and mapped accordingly.

Finally, once all elements have been parsed and instantiated, they are assembled into the Model object defined for the original transformer. This object encapsulates the full OntoUML structure and is used during the process of constraint generation. Through this modular and systematic approach, the JSONImporter provides a reliable bridge between external OntoUML representations and the internal logic of the Neo4j constraint generation pipeline.

The code for the JSON importer is available in **Appendix G**.

SECTION 3

Case Study: Uniqway, a Car-Sharing Application

3.1 Overview of Uniqway and Available Data

Our case study for an implementation of the method involves Uniqway, a car-sharing service established through a collaboration between Czech Technical University (CTU), the University of Economics in Prague (VŠE), the University of Life Sciences (CZU), ŠKODA AUTO and ŠKODA AUTO DigiLab and operated by students of the participating universties [Uni].

3.2 Data Selection and Preparation

An OntoUML model fitting the data was developed by Jiří Zikán (**Figure 3**). However, for the current study, we will work with a simplified model that represents only a subset of the data (**Figure 4**). I defined this subset by creating a simplified OntoUML model that includes primarily the entities that form the core of the problem domain and for which a reasonable number of records in the original database is available. These primarily include the following four entities with their related roles: Person-/User/Reserver/Driver, Car/ReservedCar/DrivenCar, Reservation, and Ride. To this core elements, a few additional entities have been added to allow testing of some additional features that would not have otherwise appeared.

To understand the OntoUML schema it's easier to start with the **User** class, which represents a concept that is intuitively understood in the domain. In OntoUML, `User` is stereotyped as a *role*, indicating that it is a context-dependent instantiation of a more general category—in this case, `Person`. The `User` role is mediated by a *relator* class called `Registration`, meaning that an individual becomes a user through the act of registration, which establishes a dependency between the `User` and other related entities such as an `Account`.

The `Person` class is stereotyped as a *kind*, representing a substantial entity that provides an identity criterion and persists through all its role instantiations. It defines the essential properties of a person, independently of contextual roles. While, in this schema, a `Person` only instantiates the `User` role, the separation of these classes allows for the future inclusion of other person-related roles (e.g. relate to employees), thus ensuring the model's extensibility and semantic clarity.

Each `Person` is related to one or two instances of the *quality* class `HomeAddress`, capturing address information with more precision. The use of the `quality` stereotype indicates that `HomeAddress` represents a non-substantial property of a `Person`, modeled as a separate class to support constraints (e.g., cardinality) and the possibility of attaching additional attributes (e.g., `isPrimaryAddress`). The cardinality [1..2] allows a person to have both a residential and a correspondence address.

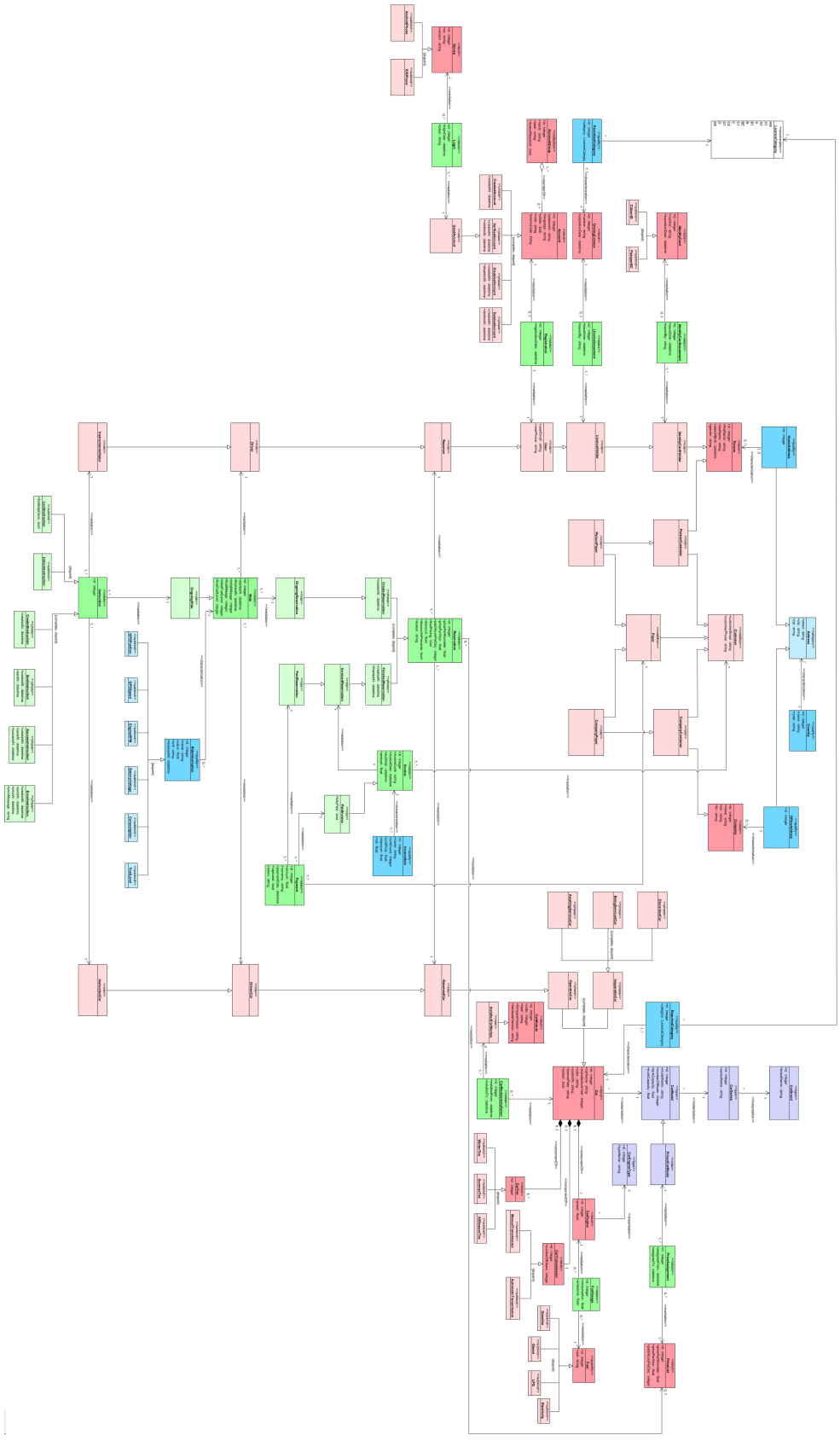

Figure 3: Original OntoUML model for Uniqway

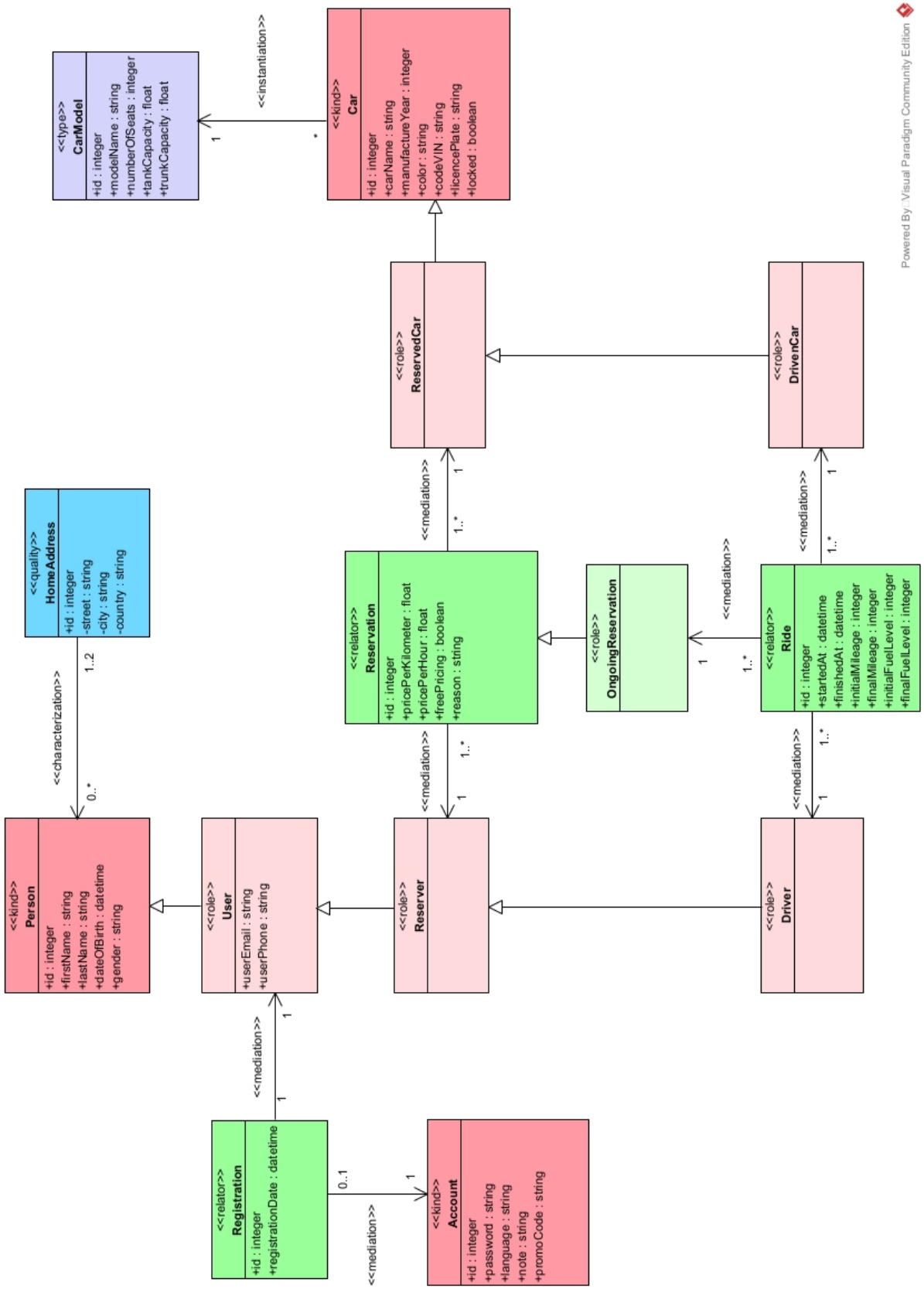

Figure 4: Simplified OntoUML model for Uniqway

The `Registration` relator connects the `User` to their `Account`, establishing the foundational context for their participation in the system. Furthermore, a `User` may play the role of a `Reserver`, and a `Reserver` may subsequently play the role of a `Rider`, depending on their participation in the `Reservation` and `Ride` relators, respectively. This role chain reflects a progression of user interactions within the system, preserving ontological rigor by distinguishing each context-dependent state.

A similar structure is adopted for vehicles. The class `Car` is stereotyped as a *kind*, representing physical cars in the domain. In way that resemble the hierarchy of `User` subtypes, it can instantiate the roles `ReservedCar` and `DrivenCar`, depending on its involvement in a `Reservation` or a `Ride`. Each `Car` is an instantiation of a `CarModel`, which is stereotyped as a *type*. In OntoUML, the *type* stereotype represents a general concept (or category) that classifies individuals without providing an identity criterion. In this case, `CarModel` provides shared structural and behavioral characteristics (e.g., brand, tank capacity, number of seats) that are common across multiple `Car` instances.

Finally, the model includes the `OngoingReservation` role, played by a `Reservation` in the context of a `Ride`. This allows the system to distinguish reservations that are actively engaged in a ride from those that are not, supporting a fine-grained representation of temporal states within the reservation lifecycle.

3.3 Data Integration into Neo4j

Executing the constraint generator with JSON export from Visual Paradigm produces the triggers to use on our Neo4j database. Since the triggers are very numerous and the script generated for their instantiation is over 2000 lines long, only the `comment.txt` file produced by the constraint generator is included in **Appendix A**. This file describes the constraints in concise human-readable form. Here, we present a simple example of one such comment and the corresponding triggers it references.
For instance, the comment `RELATIONSHIP_MUST_BE_PRESENT(RESERVER, RESERVATION)` refers to the following triggers:

```
CALL apoc.trigger.add(
"reserver_must_be_in_relationship_with_reservation_labelassigned",
'UNWIND (
    apoc.trigger.nodesByLabel($assignedLabels, "Reserver")
) AS node1
CALL apoc.util.validate(
    SIZE([(node1)--(n2:Reservation) | n2 ]) < 1,
    "Reserver label must be in relationship with Reservation label",
    null
)
RETURN null', { phase:"before" });
CALL apoc.trigger.add(
"reserver_must_be_in_relationship_with_reservation_labelremoved",
'UNWIND (
    apoc.trigger.nodesByLabel($removedLabels, "Reservation")
) AS node2
UNWIND (
    [(n1:Reserver)--(node2) | n1 ]
) AS node1
CALL apoc.util.validate(
    SIZE([(node1)--(n2:Reservation) | n2 ]) < 1,
    "Reserver label must be in relationship with Reservation label",
    null
)
RETURN null', { phase:"before" });
CALL apoc.trigger.add(
"reserver_must_be_in_relationship_with_reservation_relationdeleted",
'UNWIND (
    $deletedRelationships
) AS rel
UNWIND (
    [apoc.rel.startNode(rel), apoc.rel.endNode(rel)]
) AS node1
CALL apoc.util.validate(
    apoc.label.exists(node1, "Reserver") and
    (SIZE([(node1)--(n2:Reservation) | n2 ]) < 1),
    "Reserver label must be in relationship with Reservation label",
    null
)
RETURN null', { phase:"before" });
```

Listing 2: Script for constraint RELATIONSHIP_MUST_BE_PRESENT(RESERVER, RESERVATION)

The CSV files corresponding to the relevant tables are exported from the original PostgreSQL server and subsequently loaded on the Neo4j database through the Cypher script provided in **Appendix D**.

A major concern during the development of this script was to ensure that its memory requirements wouldn't exceed the 2GB of heap space allocated the Java Virtual Machine running the Neo4j server. This is complicated by the extensive use of triggers and the large number of tuples in some tables, such as reservations and rides (see **Appendix B** for the sizes of the various tables used). To achieve this, the script makes use of `USING PERIODIC COMMIT 1000`, which instructs Neo4j to commit every 1000 lines processed [Cyp]. This periodic commits allows the freeing up of memory space by preventing the accumulation of all intermediate changes in memory at once. However, one caveat of using periodic commits is that the commit when not all the operations for a certain query are completed, setting off triggers which would otherwise be respected. This implementation addresses this problem by assigning temporary labels (ex. *Reservation_Importing*), which are not related to any trigger, to the imported nodes. And only after the loading of the CSV is completed, a lighter query replace the real labels for the node in question and the ones related to it, at which point trigger constraints are properly enforced.

SECTION 4

Benchmark Design and Implementation

To evaluate the performance impact of triggers and verify their correct functioning, we designed a series of benchmarks that test both positive and negative scenarios. We will measure the performance with 5 positive benchmarks and 5 negative. All benchmarks used involve creation or modification of data, as this is the only case in which the constraints play a role in the performance.

Positive benchmarks measure the performance of typical database operations with and without triggers enabled. We defined five positive benchmarks to compare the execution time and assess any performance overhead introduced by the triggers.

Negative benchmarks are designed to ensure that triggers are working as intended and that their mechanisms for enforcing constraints or preventing invalid operations are effective. Again, we defined five such benchmarks to evaluate the correctness and efficiency of the triggers.

To investigate how database size affects performance, we tested the benchmarks on five different datasets of varying sizes. The size of the database was controlled by limiting the number of people loaded, with the population script loading only the reservations and rides related to these people. The sizes of the databases are available in **Appendix C**, but they correspond to total sizes that is roughly 100%, 70%, 40%, 25% and 13% of the original database and will henceforth be named DB100, DB70, DB40, DB25, DB13.

The execution plans for the benchmark queries were generated using the EXPLAIN keyword in the Neo4j browser client. An execution plan describes the steps that the Neo4j query engine will take to execute a query, including details about how data is accessed, matched, filtered, and returned. It helps to understand the query's performance characteristics and identify potential bottlenecks.

We generated execution plans for each of the different database sizes (DB100, DB70, DB40, DB25, and DB13) to check for any variations that might arise due to database size. However, since the execution plans were identical across all sizes, indicating that the query engine uses the same plan regardless of dataset scale, we will present only one representative execution plan in this report.

4.1 Positive tests

Positive tests are designed to verify that a system behaves correctly under expected conditions. In our case, we use positive tests to compare the performance of the server with constraints enabled versus the server without constraints. This allows us to measure the impact of constraints on performance in a controlled, functional scenario where all operations are valid.

For simplicity in running the tests, the positive benchmark queries are written as templates presenting variables as `${number}` and are presented as such in the following sections. This will allow us to run the same query multiple times, modifying the ids through code to respect constraints.

4.1.1 Test P1: adding new User

Test P1 involves the creation of a new user in the database. To meet the requirements, the node is created with both the `Person` and `User` labels. Additionally, the associated `HomeAddress` and Account nodes are created and the relations to them are set, together with all the necessary properties in the right type. The code for the test is provided below:

```
CREATE (p:Person {id: toInteger(${number})})
CREATE (add:HomeAddress {id: toInteger(${number})})
CREATE (acc: Account {id: toInteger(${number})})
CREATE (p)-[:LIVES_IN]->(add)
CREATE (p)-[:HAS_CORRESPONDANCE_ADDRESS]->(add)
CREATE (p) -[:REGISTERED]-> (r:Registration {id:
 ↪  toInteger(${number}), registrationDate: date("2025-05-06")})
CREATE (r)-[:HAS_ACCOUNT]->(acc)
SET p.firstName = "Jan",
p.lastName = "Novak",
p.dateOfBirth = date("1999-01-01"),
p.gender = "m",
p.userPhone = "1234567890",
add.city = "Praha",
add.country = "Czechia",
add.street = "Namesti Miru",
acc.language = "en",
acc.note = "null",
acc.password = "password",
acc.promoCode = "null",
p.userEmail = "jan.novak@gmail.com",
p:User;
```

Listing 3: Test P1 query

Figure 15 shows the corresponding execution plan. The plan for this query is simple as it uses only simple operators and it replicates 1 by 1 the CYPHER code for the query. It first executes a create operation executing all the `Create` arguments in the query, then we see `SetProperties` (and `SetProperty`) (**Figure 5**) appearing separately for each object and setting the properties in the exact order used in the query and `SetLabel` similarly setting the label User for the node p representing the person.

In the end of the query we see the generation of the result of the query, since in all the benchmark queries developed for the current thesis no return statement has been used it will always appear the same: showing an `EmptyResult`, which discards all the rows leaving nothing as a result, and `ProduceResults` which would prepare the result to be returned but in this case does nothing as it receives from `EmptyResult` (**Figure 6**).

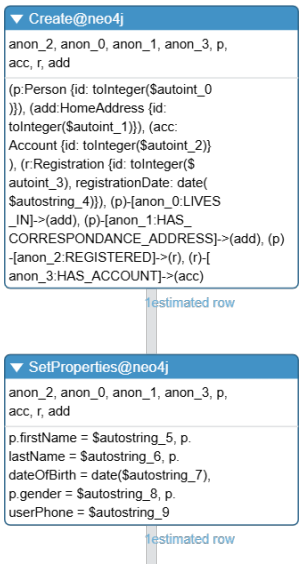

Figure 5: Zoom on element creation and setting of properties

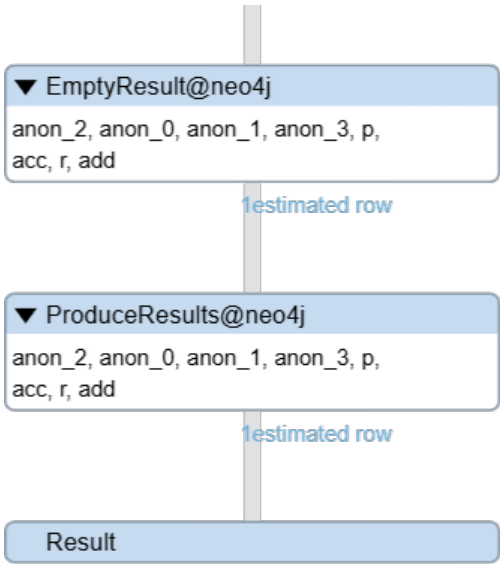

Figure 6: Zoom on element empty result production

Interestingly the execution plan shows the exact same structure used in the query. Which means that the query optimizer didn't unite the setting of the properties from the same object or integrating the setting of properties in the creation of the nodes. This result is relevant as it shows that either there is no significant performance difference between the setting of properties in the Create operator, in `SetProperties` or in `SetProperty` or that the syntactical choice in the writing of the Cypher query plays a very significant role on the performance of the query once executed.

4.1.2 Test P2: adding a new reservation

Test P2 involves the creation of a new reservation for a User and a Car, it additionally sets the participants as `Reserver` and `ReservedCar` to ensure compliance with constraints. The code for the test is provided below:

```
MATCH (u:User {id: toInteger(123)}), (c:Car {id: toInteger(51)})
CREATE (r:Reservation  {id: toInteger(${number}), pricePerKilometer
 ↪  : 23.5, pricePerHour : 31.2, freePricing : toBoolean("true"),
 ↪  reason: "null"})
CREATE (u)-[:RESERVED]->(r)
CREATE (c)-[:IS_RESERVED]->(r)
SET r:Reservation,
u:Reserver,
c:ReservedCar;
```

Listing 4: Test P2 query

Figure 16 shows the execution plan for this query. It starts by searching the User and the Car nodes by the id. An important thing to note from this query is that we can see it leverages the constraint on the uniqueness of the id to find the Car, since this constraint creates an index allowing Neo4j to use the `NodeUniqueIndexSeek` operator, but it cannot do the same on User, being forced to use a `NodeByLabelScan` on the label `User` followed by a filter on the value of the `id` property (**Figure 7**). This happens because nowhere has a constraint on uniqueness been set for the id of User, even though we know it must be unique due to id being part of the properties defined by Person, a superclass of user, for which it has been defined as unique. To compare the performance I propose an alternative version of TestP2, called Test P2Alt.

After the fetching operators we see that the results from both branches are combined in a rows composed by the `User` node and the `Car` node, with the `CartesianProduct` operator. In this case the combination of the rows is trivial as we receive one row only from each branch, but if it receives multiple rows from the two banches the operator will combine each row from one branch with every single row from the other branche, as the name implies.

The execution plane then continues as with the same nodes that appeared in the previous query matching closely the Cypher query and returning no result.

4.1.3 Test P2Alt: creating a reservation leveraging the index

Test P2Alt substitutes the User label with Person to leverage the efficiency provided by the uniqueness constraints on the id, which makes the database generate an index on it. The code follows below:

```
MATCH (u:Person {id: toInteger(123)}), (c:Car {id: toInteger(51)})
CREATE (r:Reservation  {id: toInteger(${number}), pricePerKilometer
 ↪  : 23.5, pricePerHour : 31.2, freePricing : toBoolean("true"),
 ↪  reason: "null"})
CREATE (u)-[:RESERVED]->(r)
CREATE (c)-[:IS_RESERVED]->(r)
SET r:Reservation,
u:Reserver,
c:ReservedCar;
```

Listing 5: Test P2alt query

Figure 17 shows the execution plan where we can see the initial `NodeByLabelScan+Filter` to match the User element has been substituted by Index seek and united with the similar operation used for the Car element into a `MultiNodeIndexSeek` ((**Figure 8**)). This operator can retrieve nodes from both MATCH statements and combine them with a cartesian product. Again in this case the cartesian product is trivial as both indexes have the UNIQUE constraints, but Neo4j supports also non unique indexes in which case this operator can also be used.

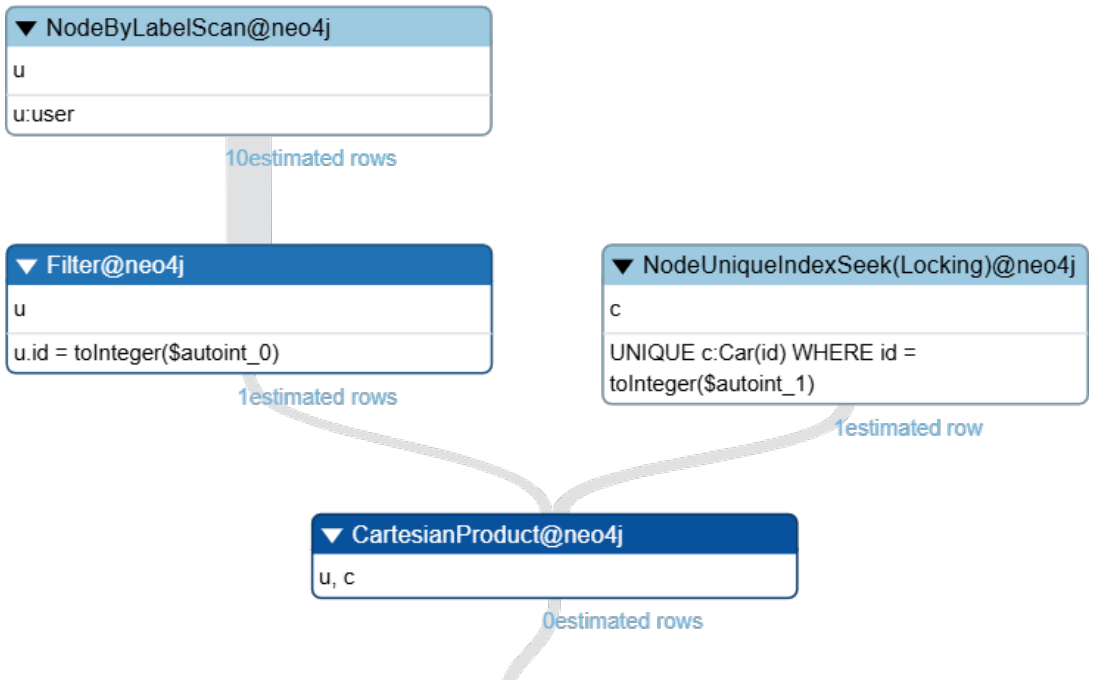

Figure 7: Zoom on matching operation for P2

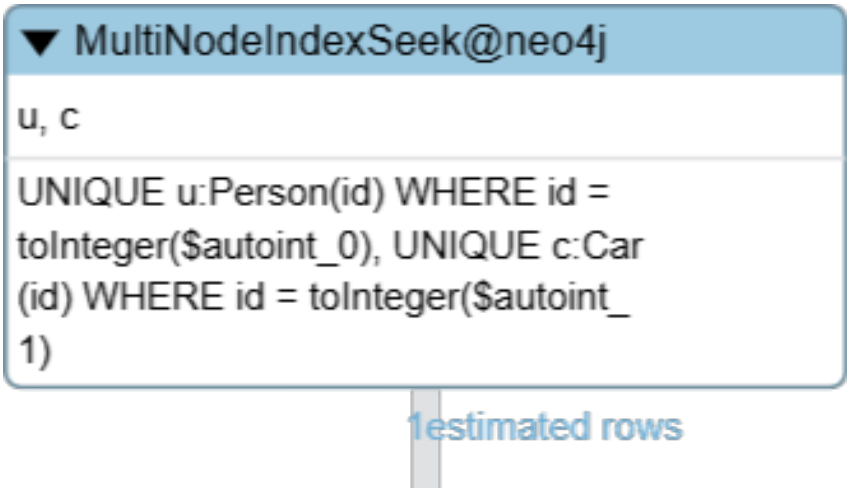

Figure 8: Zoom on matching operation for P2alt

4.1.4 Test P3: Creating a new car

Test P3 is a straightforward query that creates a new car node and associates it an already existing car model.

```
 MATCH (cm:CarModel {id: toInteger(4)})
CREATE (n:Car)
CREATE (n)-[r:IS_OF_MODEL]->(cm)
SET n.id = toInteger(${number}),
n.carName = "Ludmila",
n.manufactureYear = toInteger("2025"),
n.color = "#F00",
n.codeVIN = "TMBGR9NW1L3034800",
n.licencePlate = "2ME41423",
n.locked = toBoolean("false")
```

Listing 6: Test P3 query

Figure 18 shows the execution plan for this query. As we can see it is very straightforward as the `CarModel` node is fetched with `NodeUniqueIndexSeek` and then the Cypher query is closely replicated with the node we already saw.

4.1.5 Test P4: Creating a new ride

Test P4 creates a new ride. It matches 1 single Reservation that doesn't bear the label OngoingReservation, implying it doesn't have an associated ride, and associates the new ride with it and its associated user and car.

```
MATCH (res:Reservation)
WHERE NOT res:OngoingReservation
WITH res LIMIT 1
MATCH (u:User)-[:RESERVED]->(res), (c:Car)-[:IS_RESERVED]->(res)
CREATE (rr:Ride  {id: toInteger(${number}),   startedAt :
↪  date("2025-01-01"),
    finishedAt : date("2025-01-01"), initialMileage :
    ↪  toInteger(418),
    finalMileage: toInteger(444),
    initialFuelLevel: toInteger(75),
    finalFuelLevel: toInteger(60)})
CREATE (rr)-[:FROM_RESERVATION]->(res)
CREATE (u)-[:RIDES]->(rr)
CREATE (c)-[:IS_DRIVEN]->(rr)
SET res:OngoingReservation,
u:Driver,
c:DrivenCar;
```

Listing 7: Test P4 query

Figure 19 shows the plan for this query. As can see it gets all the nodes labeled as `Reservation`, filters out all those that bear the label `OngoingReservation`, then the operator `Limit` which returns only the first n Lines of the input it receives, in this case 1. The operator `Eager` serves as a way to sinchronize the pipeline, forcing all the previous operators to execute fully before continuing execution. The operator visualizations even informs us as to why this operator has been inserted. In this case we read "overlapping set labels: `OngoingReservation`", meaning that the next operators change the label `OngoingReservation`, potentially influencing the previous `Filter` operator. By analyzing carefully the execution plan we notice that it is indeed superfluous as it only modifies the label for the node that has already been received, in addition limiting the rows to one means that when the `Eager` operator is reached the first time the execution of the previous operators is already complete, as all the other rows are discarded (**Figure 9**). [Cyp]

The rest of the query follows closely the Cypher query using operators that already appeared in the previous queries with the exception of `Expand(All)` operator which is used to traverse relationships and retrieve the related nodes (**Figure 10**).

Special attention is to be given to the fact that if we specify the Label for the object we are retrieving from the relation Neo4j will have to use a filter to verify that label, even when we already know thanks to our predefined model that all nodes we can reach through that relationship bear that label. As a consequence this should be considered when optimizing query for small improvement which could still be meaningful when applied to massively executed queries.

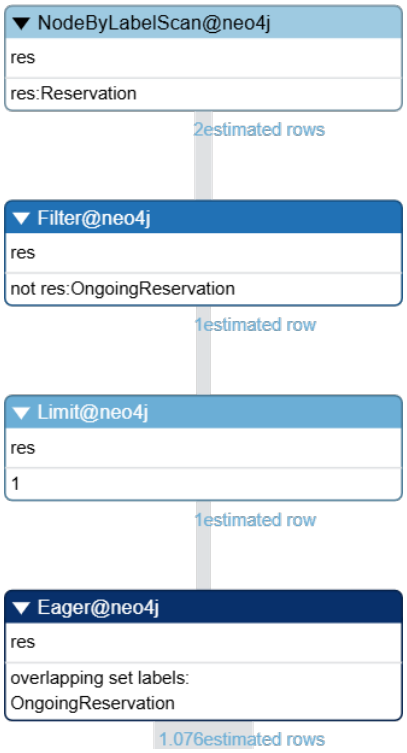

Figure 9: Zoom on the eager operator

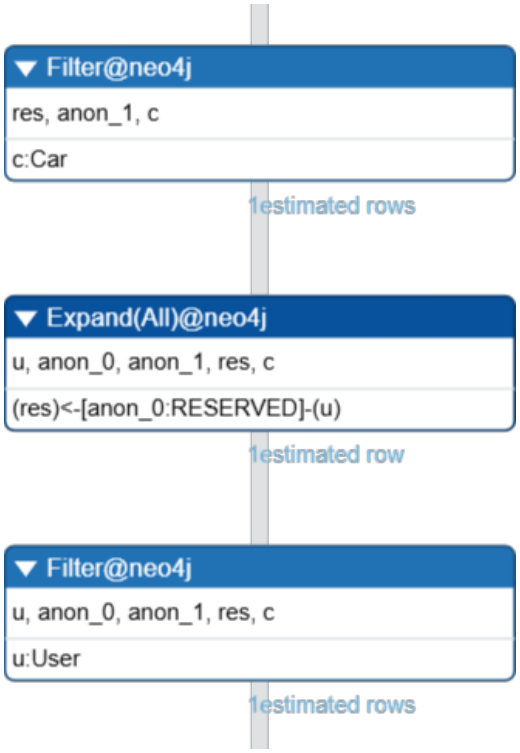

Figure 10: Zoom on the Expand(All) operator

4.1.6 Test P5: Changing the address of a person

Test P5 involves changing the address of a person. The writing of such a query in a form compliant with the constraints is complex as multiple operations have to be done before the triggers are checked. The implementation used here looks for the person and its LIVES_IN relationship which ties it to the home address (there is also another relationship to HomeAddress which shows the correspondance address), creates the new address, deletes the relationship with the previous address and substitutes it with a relationship to the new address, then a subquery follows enclosed in a CALL { ... } statement which checks if the old addressed is still holding a relationship (in case more than one person have the some address or the person still holds that address as the correspondence address) to delete only in case it doesn't hold any. The use of the OPTIONAL MATCH is necessary since it doesn't discard the the rows for which the match produces no result, allowng the continuation of the execution.

```
MATCH (p:Person {id: 50})-[r1:LIVES_IN]->(a1)
CREATE (a2:HomeAddress {
  id: toInteger(${number}),
  city: "Praha",
  country: "Czechia",
  street: "Namesti Miru"
})
DELETE r1
CREATE (p)-[:LIVES_IN]->(a2)
WITH a1
CALL {
  OPTIONAL MATCH (a1)--()
  WITH a1, count(*) AS relCount
  WHERE relCount = 0
  DELETE a1
}
```

Listing 8: Test P5 query

Figure 20 shows the plan for this query. The visualization is ambiguous because of how it shows the execution of the subquery but by analyzing it and comparing it with Cypher code we can notice how the branch on the left is the first part of the query, after it has created the new address and subistuted the relation to the old address with a relation between the person and the newly created adress it reches a node `SubqueryForeach` which executes the other branch, the callee, for each row it receives (**Figure 11**).

The first operator is `Argument` which receives the variables from the Caller and constitutes the start of the subquery. Then the execution continues using `OptionalExpand(all)` for the expand match on the relations and `EagerAggregation` for the calculation of the relCount aggregate variable.

4.2 Negative tests

Negative tests, on the other hand, are used to ensure the system correctly handles invalid or unexpected inputs. Here, we use negative tests to verify that the defined constraints are enforced properly. These tests deliberately attempt to violate the constraints, and we check that the server appropriately rejects or handles these operations.

4.2.1 Test N1: Creating a new ride without defining a starting time

Test N1 involves a query creating a new `Ride` node respecting all the constraints except that is doesn't set the property `startedAt` which is specified as a mandatory property in the OntoUML for the class Ride.

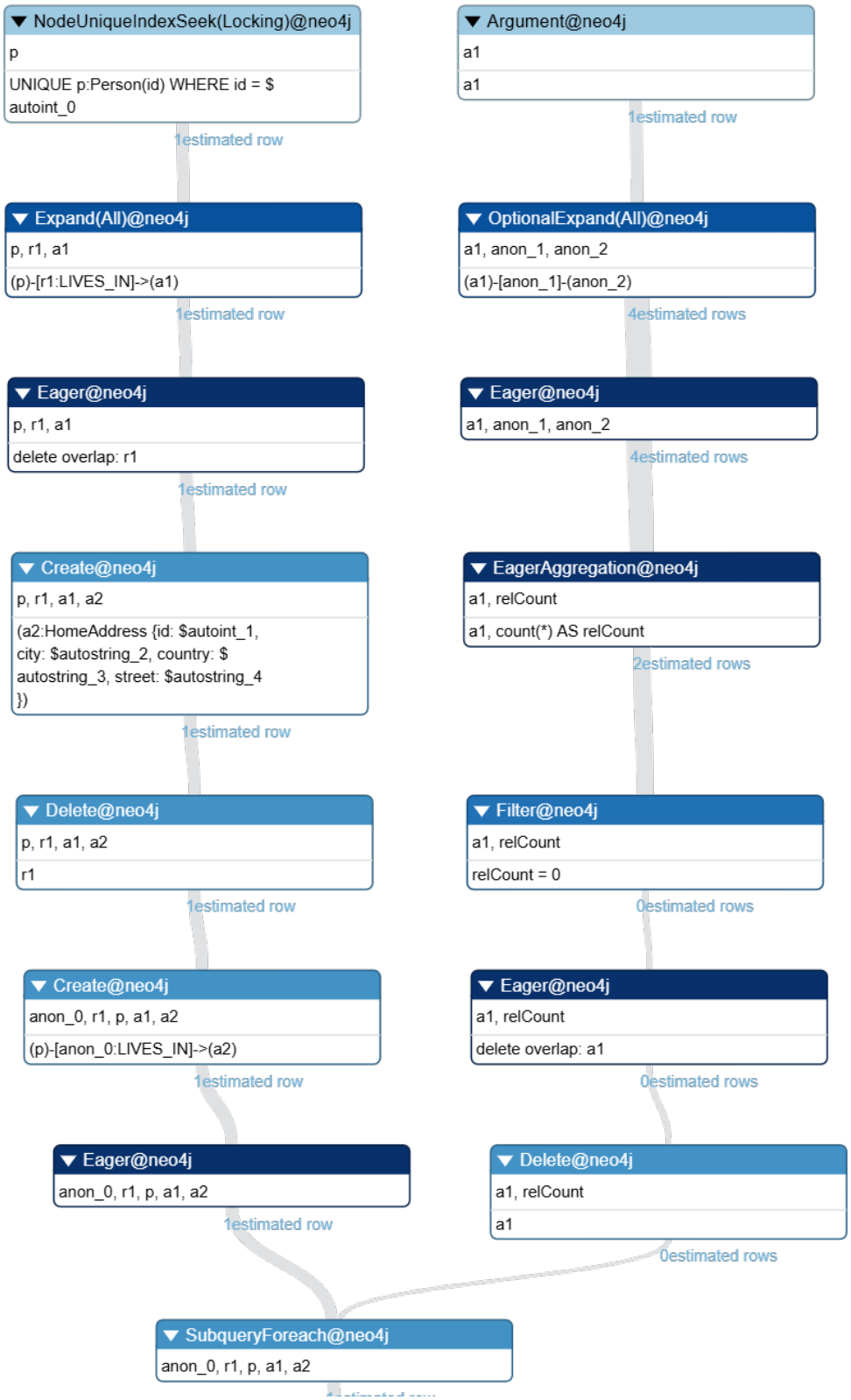

Figure 11: Zoom on the execution of SubqueryForeach

```
    MATCH (res: Reservation {id: toInteger(7841)}),
    (u:User)-[:RESERVED]->(res), (c:Car)-[:IS_RESERVED]->(res)
    CREATE (rr:Ride  {id: toInteger(21344556),
        finishedAt : date("2025-01-01"), initialMileage :
        ↪  toInteger(418),
        finalMileage: toInteger(444),
        initialFuelLevel: toInteger(75),
        finalFuelLevel: toInteger(60)})
    CREATE (rr)-[:FROM_RESERVATION]->(res)
    CREATE (u)-[:RIDES]->(rr)
    CREATE (c)-[:IS_DRIVEN]->(rr)
    SET res:OngoingReservation,
    u:Driver,
    c:DrivenCar;
```

Listing 9: Test N1 query

From the file `comments.txt` (available in Appendix A)) we already know of the existence of the constraint `PROPERTY_MUST_BE_PRESENT(Ride, startedAt)`. The definition of this trigger can be recovered from the `triggers.cql` file generated by the transformer and is the following:

```
//PROPERTY_MUST_BE_PRESENT(RIDE, STARTEDAT)
CALL apoc.trigger.add("ride_startedat_must_be_present",
'UNWIND (
    $createdNodes +
    apoc.trigger.nodesByLabel($assignedLabels, "Ride") +
    apoc.trigger.nodesByLabel($removedNodeProperties, "Ride")
) AS node
CALL apoc.util.validate(
    apoc.label.exists(node, "Ride") and not
    exists(node.startedAt),
    "startedAt property of Ride must be present",
    null
)
RETURN null', { phase:"before" });
```

Listing 10: Script for constraint PROPERTY_MUST_BE_PRESENT(Ride, startedAt)

Which means that whenever a node is created, the label `Ride` of node is added or removed, or the properties of a node labeled as `Ride` has been changed the trigger

throws an error if the label `Ride` is present and the property `startedAt` is absent.

Figure 21 shows the plan for this query.

4.2.2 Test N2: Creating a new car without a model

Test N2 involves a query that attempts the creation of a `Car` node without creating a relation from it to its relative `CarModel` node. This should be an error as the OntoUML model specifies the relationship between the classes Car and CarModel as ″*–1″ making it mandatory for car to have exactly one relation to `CarModel`.

```
CREATE (n:Car)
SET n.id = toInteger(9949),
n.carName = "Ludmila",
n.manufactureYear = toInteger("2025"),
n.color = "#F00",
n.codeVIN = "TMBGR9NW1L3034800",
n.licencePlate = "2ME41423",
n.locked = toBoolean("false")
```

Listing 11: Test N2 query

From the file `comments.txt` (available in **Appendix A**)) we know the existence of the constraints `RELATIONSHIP_MUST_BE_PRESENT(Car, CarModel)` and `RELATIONSHIP_MUST_BE_LIMITED(Car, CarModel, 1)` which ensure that the relationship between `Car` and `CarModel` respects the model. In particular the current test doesn't respect `RELATIONSHIP_MUST_BE_PRESENT(Car, CarModel)` which is created by the following command:

```
//RELATIONSHIP_MUST_BE_PRESENT(CAR, CARMODEL)
CALL apoc.trigger.add(
'car_must_be_in_relationship_with_carmodel_labelassigned',
'UNWIND (
    apoc.trigger.nodesByLabel($assignedLabels, "Car")
) AS node1
CALL apoc.util.validate(
    SIZE([(node1)--(n2:CarModel) | n2 ]) < 1,
    "Car label must be in relationship with CarModel label",
    null
)
RETURN null', { phase:"before" });
CALL apoc.trigger.add(
'car_must_be_in_relationship_with_carmodel_labelremoved',
'UNWIND (
```

```
16      apoc.trigger.nodesByLabel($removedLabels, "CarModel")
17  ) AS node2
18  UNWIND (
19      [(n1:Car)--(node2) | n1 ]
20  ) AS node1
21  CALL apoc.util.validate(
22      SIZE([(node1)--(n2:CarModel) | n2 ]) < 1,
23      "Car label must be in relationship with CarModel label",
24      null
25  )
26  RETURN null', { phase:"before" });
27  CALL apoc.trigger.add(
28  "car_must_be_in_relationship_with_carmodel_relationdeleted",
29  'UNWIND (
30      $deletedRelationships
31  ) AS rel
32  UNWIND (
33      [apoc.rel.startNode(rel), apoc.rel.endNode(rel)]
34  ) AS node1
35  CALL apoc.util.validate(
36      apoc.label.exists(node1, "Car") and
37      (SIZE([(node1)--(n2:CarModel) | n2 ]) < 1),
38      "Car label must be in relationship with CarModel label",
39      null
40  )
41  RETURN null', { phase:"before" });
```

Listing 12: Script for RELATIONSHIP_MUST_BE_PRESENT(Car, CarModel)

As we can see it is maintained by three different triggers. In the first one, validation is run for all nodes to which the label `Car` is added and it involves the calculation of the size of a list comprehension, throwing an error if the size is less than 1. The list comprehension means [(node1)–(n2:CarModel) — n2] means that it finds all relations matching the pattern represented on the left side (relations between the triggering node and `CarModel`-labeled nodes) and returns a list of the term represented on the right side, i.e. a list of all the the `CarModel` nodes related to the triggering node.

The other two scripts involves a similar check when a label `CarModel` is removed, possibly causing related cars to lose the related `Car` nodes to lose their `CarModel`, and when a relationship is deleted. In this last case the trigger checks for each extremity of the relation if it is a `Car` and if it is still related to `CarModel` node.

Figure 22 shows the plan for this query.

4.2.3 Test N3: Creating a new ride without it being related to a reservation

Test N3 involves the creation of a new `Ride` node without it being related to an `OngoingReservation` node while respecting all the other constraints. The relationship between `Ride` and `OngoingReservation` is defined in the OntoUML model as "1..*–1", which means that each `Ride` must be related to exactly one `OngoingReservation` and an `OngoingReservation` must be related to at least one `Ride`.

```
MATCH (u:User {id: toInteger(8519)}), (c:Car{id: toInteger(60)})
CREATE (rr:Ride  {id: toInteger(21345564),
    startedAt : date("2025-01-01"),
    finishedAt : date("2025-01-01"), initialMileage :
     ↪  toInteger(418),
    finalMileage: toInteger(444),
    initialFuelLevel: toInteger(75),
    finalFuelLevel: toInteger(60)})
CREATE (u)-[:RIDES]->(rr)
CREATE (c)-[:IS_DRIVEN]->(rr)
SET  u:Driver,
c:DrivenCar;
```

Listing 13: Test N3 query

The following constraints are responsible of ensuring that the relationship rules are respected as specified by the OntoUML model:
`RELATIONSHIP_MUST_BE_PRESENT(OngoingReservation, Ride)`,
`RELATIONSHIP_MUST_BE_LIMITED(Ride, OngoingReservation, 1)` and
`RELATIONSHIP_MUST_BE_PRESENT(Ride, OngoingReservation)`.
In this case the constraint violated is the latter which is defined in the following way:

```
//RELATIONSHIP_MUST_BE_PRESENT(RIDE, ONGOINGRESERVATION)
CALL apoc.trigger.add(
"ride_must_be_in_relationship_with_ongoingreservation_labelassigned",
'UNWIND (
    apoc.trigger.nodesByLabel($assignedLabels, "Ride")
) AS node1
CALL apoc.util.validate(
    SIZE([(node1)--(n2:OngoingReservation) | n2 ]) < 1,
    "Ride label must be in relationship with OngoingReservation
     ↪  label",
    null
)
RETURN null', { phase:"before" });
```

```
CALL apoc.trigger.add(
"ride_must_be_in_relationship_with_ongoingreservation_labelremoved",
'UNWIND (
    apoc.trigger.nodesByLabel($removedLabels, "OngoingReservation")
) AS node2
UNWIND (
    [(n1:Ride)--(node2) | n1 ]
) AS node1
CALL apoc.util.validate(
    SIZE([(node1)--(n2:OngoingReservation) | n2 ]) < 1,
    "Ride label must be in relationship with OngoingReservation
     ↪ label",
    null
)
RETURN null', { phase:"before" });
CALL apoc.trigger.add(
"ride_must_be_in_relationship_with_ongoingreservation_relationdeleted",
'UNWIND (
    $deletedRelationships
) AS rel
UNWIND (
    [apoc.rel.startNode(rel), apoc.rel.endNode(rel)]
) AS node1
CALL apoc.util.validate(
    apoc.label.exists(node1, "Ride") and
    (SIZE([(node1)--(n2:OngoingReservation) | n2 ]) < 1),
    "Ride label must be in relationship with OngoingReservation
     ↪ label",
    null
)
RETURN null', { phase:"before" });
```

Listing 14: Script for RELATIONSHIP_MUST_BE_PRESENT(Ride, OngoingReservation)

As this constraint is of the type `RELATIONSHIP_MUST_BE_PRESENT`, appearing also in Test N2, we can compare the two and notice how the implementation is exactly the same. With just a simple change in the labels involved.

Figure 23 shows the plan for this query. As we can see, we again see the situation appeared in Test N2 because we are searching a `User` node by `id`. I therefore also include a new TestN3alt which uses `Person` instead of `User`, similar to how it happened earlier (**Figure 12**).

4.2.4 Test N3alt

```
MATCH (u:Person {id: toInteger(8519)}), (c:Car{id: toInteger(60)})
CREATE (rr:Ride  {id: toInteger(21345564),
    startedAt : date("2025-01-01"),
    finishedAt : date("2025-01-01"), initialMileage :
    ↪  toInteger(418),
    finalMileage: toInteger(444),
    initialFuelLevel: toInteger(75),
    finalFuelLevel: toInteger(60)})
CREATE (u)-[:RIDES]->(rr)
CREATE (c)-[:IS_DRIVEN]->(rr)
SET  u:Driver,
c:DrivenCar;
```

Listing 15: Test N3alt query

As long as the `Person` node we are using is also a `User` the considerations in terms of constraints involved are exactly the same as earlier.

Figure 24 shows the plan for this query, which shows the usage of `MultiNodeIndexSeek` we were aiming for (**Figure 13**).

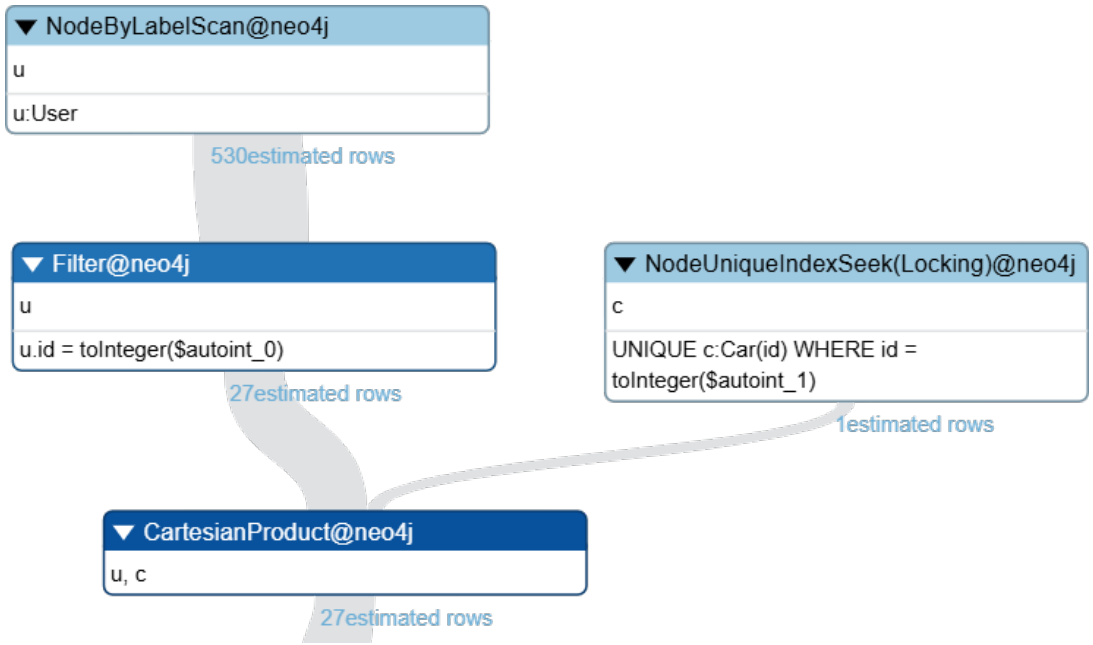

Figure 12: Zoom on matching operation for N3

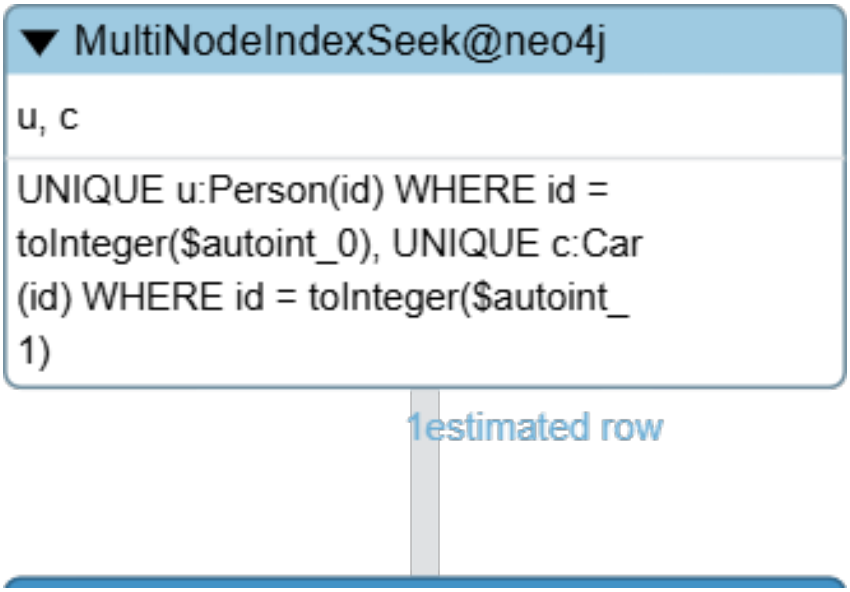

Figure 13: Zoom on matching operation for N3alt

4.2.5 Test N4: Creating a new reservation without associating a car

Test N4 involves creating a new `Reservation` node without associating a `ReservedCar` node. This violates the definition of the relation between `Reservation` and `ReservedCar`.

```
MATCH (u:User {id: toInteger(123)})
CREATE (r:Reservation  {id: toInteger(99959999),
pricePerKilometer : 23.5, pricePerHour : 31.2,
freePricing : toBoolean("false"), reason: "null"})
CREATE (u)-[:RESERVED]->(r)
SET r:Reservation,
u:Reserver;
```

Listing 16: Test N4 query

Figure 25 shows the plan for this query.

4.2.6 Test N5: Creating a new reservation by an unregistered person

Test N5 involves the creation of an unregistered `Person`, that doesn't have an associated `Registration` and doesn't hold the label `User`, and of a new `Reservation` where this `Person` appears as Reserver. This shouldn't be possible as the OntoUML model (**Figure 4**) shows that `Reserver` inherits from `User` and therefore must hold this label too.

```
MATCH (c: Car {id:60})
CREATE (p:Person {id: toInteger(9999959)})
```

```
CREATE (a:HomeAddress {id: toInteger(23782342)})
CREATE (p)-[:LIVES_IN]->(a)
CREATE (p)-[:HAS_CORRESPONDANCE_ADDRESS]->(a4)
CREATE (r:Reservation  {id: toInteger(99999979),
    pricePerKilometer : 23.5, pricePerHour : 31.2,
    freePricing : toBoolean("true"), reason: "null"})
CREATE (p)-[:RESERVED]->(r)
CREATE (c)-[:IS_RESERVED]->(r)
SET p.firstName = "Jan",
p.lastName = "Novak",
p.dateOfBirth = date("1999-01-01"),
p.gender = "m",
p.userPhone = "1234567890",
a.city = "Praha",
a.country = "Czechia",
a.street = "Namesti Miru",
r:Reservation,
p:Reserver,
c:ReservedCar;
```

Listing 17: Test N5 query

Looking at the constraints we can identify `LABEL_MUST_BE_IN_COMBINATION(Reserver, User)` as the constraint that ensures the generalization from `Reserver` to `User`. It is defined this way:

```
CALL apoc.trigger.add('reserver_must_be_with_user',
'UNWIND (
    apoc.trigger.nodesByLabel($assignedLabels, "Reserver") +
    apoc.trigger.nodesByLabel($removedLabels, "User")
) AS node
CALL apoc.util.validate(
    apoc.label.exists(node, "Reserver") and not
    (apoc.label.exists(node, "User")),
    "Reserver label must be in a combination with User labels",
    null
)
RETURN null', { phase:"before"});
```

Listing 18: Script for LABEL_MUST_BE_IN_COMBINATION(Reserver, User)

As we can see the triggers acts when a query assigns the label `Reserver` or removes the label `User` from a node and it throws an error when it finds a node which is a `Reserver` but not a `User`.

For the sake of proving the sufficiency of the constraints we can describes some ways we could try to go around this constraint and create `Reservation` from an unregistered `Person` and how they would fail. Since the constraint are previous query didn't comply with was `LABEL_MUST_BE_IN_COMBINATION(Reserver, User)` we can start by trying to avoid using the label Reserver. In this case we would run into the constraint `RELATIONSHIP_MUST_BE_PRESENT(Reservation, Reserver)` as it wouldn't be satisfied by a `Person` node which is not `Reserver`. Another way to avoid the constraint would be to assign the label `User` to the `Reserver`, this would trigger the constraint
`RELATIONSHIP_MUST_BE_PRESENT(User, Registration)`. This ensures that a negligent developer wouldn't be able to violate schema requirements by simply using the labels improperly.

Figure 26 shows the plan for this query.

SECTION 5

Evaluation and Results

5.1 Methodology

All benchmarks were executed on my local machine to ensure consistency in the testing environment. The specifications of the machine used for these benchmarks are provided in **Appendix F**. Since the constraint transformer is not yet compatible with Neo4j 5, all tests were carried out using Neo4j version 4.4.42. This version was selected to ensure compatibility with the current implementation of the transformer. Additionally, Neo4j 4.4.42 requires Java 11, so the database was run on OpenJDK 11.0.26.4. Newer versions of the JDK are not supported by this Neo4j release, making it necessary to use an older Java environment. This setup provides a reliable baseline for evaluating the transformer's functionality until support for Neo4j 5 is implemented.

The tests were carried out using the Neo4j Python Driver 5.28. The Neo4j Python Driver is an official client library that allows Python applications to connect to and interact with Neo4j graph databases. It provides a high-level API to execute Cypher queries, manage transactions, and handle query results. Under the hood, it communicates with the Neo4j database using the Bolt protocol, a binary protocol specifically designed by Neo4j for efficient communication between clients and the database. Bolt is optimized for low-latency, high-throughput transmission of Cypher queries and results, making it significantly faster and more efficient than traditional HTTP-based interactions. The driver handles connection pooling, authentication, and data serialization, abstracting away the complexity of the Bolt protocol while leveraging its performance benefits.

Each test was executed 1,000 times for each database size, both with and without constraints. The total execution time was recorded and then divided by 1,000 to obtain the average time in milliseconds.

Below is the code of the function used to run the tests for positive benchmarks. As previously mentioned, positive benchmarks were written as templates to allow the use of different triggers while ensuring compliance with the specified constraints. In this function, `baseint` serves as the starting number from which IDs are assigned; it is incremented by one on each iteration. To prevent duplicate IDs, a sufficiently large `baseint` value is chosen each time `run_cypher_query` is executed on the same instance, ensuring no conflicts with existing IDs in the database.

```
def run_cypher_query(basequery, baseint, user = "neo4j", password =
↪  "password"):

    driver = GraphDatabase.driver("bolt://localhost:7687",
        auth=(user, password))
    session = driver.session()
    m=1000
    start_time = time.time()
    try:
        for i in range(1, m):
            result = session.run(Template(basequery)
                .substitute(number=baseint+i))
            summary = result.consume()
    except ConstraintError as e:
        print("Contstraint error: " + e.message)
    except ClientError as e:
        print("Error: " + e.message)
    except Exception as e:
        print("Unexpected error: " + e.message)
    end_time = time.time()
    return (end_time - start_time)*1000/m
```

Listing 19: Python function running performance test on positive queries

Below is the function used to run tests for negative benchmarks. This function is similar to the positive test function, but in this case the for-loop terminates if the operation does not raise a `ConstraintError` or `ClientError`, since that indicates that the constraints are not enforced as expected and the error message is captured from the first occurrence of the expected error, enabling validation that the error is of the correct type. While the performance when failing the constraints is not as relevant as in the case of working queries as this kind of test cases display a failure in the implementation and do not represent how the working performance of the database in a normal setting, they are displayed here for completeness and to show how the database doesn't perform particularly differently when commit a transaction or executing a rollback due to constraint failure.

```
def run_negative_test(query,  user = "neo4j", password =
↪  "password"):

    driver = GraphDatabase.driver("bolt://localhost:7687",
        auth=(user, password))
    session = driver.session()
    m=1000
    start_time = time.time()
    errorMessage = ""
    first = True
    for i in range(1, m):
        trig = False
        try:
            result = session.run(query)
            summary = result.consume()
        except ConstraintError as e:
            trig = True
            if(first):
                first = False
                errorMessage = "Constraint successfully triggered: "
                    + e.message
        except ClientError as e:
            trig = True
            if(first):
                first = False
                errorMessage = "Constraint successfully triggered: "
                    + e.message
        except Exception as e:
            errorMessage = "Unexpected error: " + e.message
            break
        if(trig==False):
            errorMessage = "ERROR: No constraint triggered"
            break
    end_time = time.time()
    return errorMessage, (end_time - start_time)*1000/m
```

Listing 20: Python function running performance test on negative queries

	DB13	DB25	DB40	DB70	DB100
Without Triggers	15.157	16.649	15.207	15.035	9.954
With Triggers	49.344	47.309	43.504	50.311	80.679

Table 2: P1 execution times

	DB13	DB25	DB40	DB70	DB100
Without Triggers	8.343	9.753	9.436	11.563	11.930
With Triggers	31.421	30.512	31.177	34.256	46.203

Table 3: P2 execution times

5.2 Results and discussion

5.2.1 P1: adding new User

Table 2 shows the average time in milliseconds needed for the execution of the query relative to the database size and to presence or absence of triggers.

This first test provides us already with a great amount of information. The first we can see is that even by averaging the times from 1000 executions the results are still strongly influenced by randomness, leading to no clear correlation and a more difficult analysis. Nevertheless, a few things can be evidently noticed.

As regards the influences of triggers on the time we can see that running the query on the database without triggers consistently takes more then thrice the time needed with triggers enabled. A similar slow down was to be expected as this test creates three different nodes, one of which (`Person`) involving even a second label (`User`), triggering validations of properties, relations and labels from 4 different classes.

When considering the influence of database size the first we notice is that the results in both cases for DB100 show clear degeneracy from what would be expected looking at other results. We can evidently discard the possibility that for larger database instances we obtain better performance without triggers but the results still show that there is no clear correlation between time and database size at all. For this query this is to be expect as it involves only write operations.

If we look at the influence of size when triggers are included again the only result that shows a possible different performance is DB100, while when considering the others it seems to be roughly constant. While we accept the possibility that the performance is getting influenced, this is not necessarily so: as all the elements involved are created in the query itself none of the the triggers involved has to perform checks involving other existing elements the number of which would be related to the size of the database. In additions constraints checking on the relations do not require joins thanks to the native graph implementation of Neo4j.

5.2.2 P2 and P2alt: adding a new reservation

Table 3 and **Table 4** show respectively the average execution times recorded for **Test P2** and Test **P2alt**.

In this queries the magnitude of the penalty to the performance caused by the validation of the performance seems to be sensibly lower than that observe in **Test P1**. This observation is consistent with the fact that this query involves the creation of

	DB13	DB25	DB40	DB70	DB100
Without Triggers	10.843	13.612	10.117	11.251	14.097
With Triggers	34.210	30.537	33.314	37.145	43.537

Table 4: P2alt execution times

	DB13	DB25	DB40	DB70	DB100
S1 Without Triggers	3.883	3.977	4.305	4.863	5.958
S2 Without Triggers	3.864	6.51	4.495	4.916	6.084
S1 With Triggers	4.199	4.211	4.294	4.803	6.423
S2 With Triggers	3.557	3.331	3.385	3.418	3.583

Table 5: Simple match execution times

single node and the application of 2 labels related to *SubKinds* having no additional constraints related to them except a generalization and the relation to `Reservation`.

This time we also notice a clear correlation between performance and database size. The reason for this is that the query starts by searching 2 nodes, operation which is of course strongly dependent on size when we use a node scan followed by a filter as we do in P2 but also when we execute an index seek, although to a lesser degree. As a consequence of this it is a surprising result that we see no performance increase in P2alt when we compare its results with those obtained by P2. This could probably be due to the fact that most of the time is taken by writing to memory and constraint checking which in addition to the inevitable randomness in the results causes the difference to be obscure. A more useful test can be done by comparing the speed of the following two queries on different database sizes with and without constraints.

```
1 MATCH (u:User {id:
  ↪ toInteger(123)})
2 RETURN u
```

Listing 21: S1 : Matching with no index

```
1 MATCH (u:Person {id:
  ↪ toInteger(123)})
2 RETURN u
```

Listing 22: S2 : Matching with an index

Table 5 shows the results by referring to the match on User as S1 and the match on Person as S2. As we can see the tests that make use of scan + filter (both cases without triggers and S1 with triggers) show a clear dependence on the size of the database while the query that makes use of the index seek operation is very efficient and the delays correlated with database size appear minimal. This can bettere be seen with the graph in **Image 14**

5.2.3 P3: Creating a new car

Table 6 shows the results from **Test P3**. The results lead to same conclusions reached about **Test P1**, with constant time when working without triggers relative to time, and

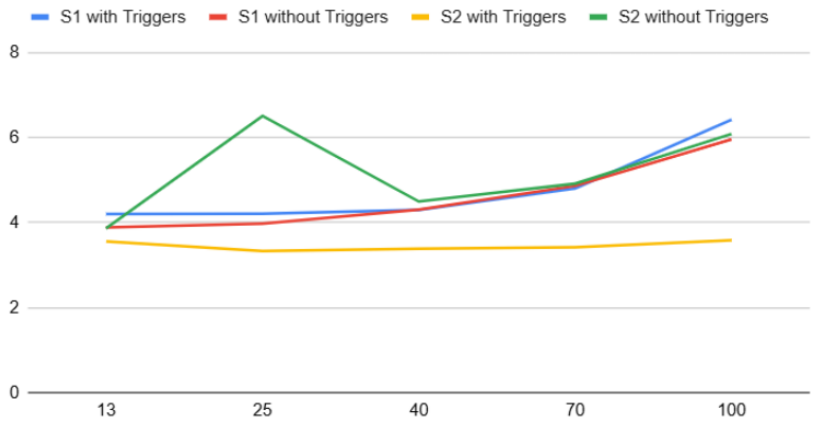

Figure 14: Graph of the match queries execution times

	DB13	DB25	DB40	DB70	DB100
Without Triggers	9.661	10.964	9.496	9.966	10.465
With Triggers	28.100	27.710	26.840	56.695	33.438

Table 6: P3 execution times

a constant increase in time with strong fluctuations when working with triggers.

5.2.4 P4: Creating a new ride

The results from the time measurements of **Test P4** can be found on **Table 7**. **Table 7** shows the results for **Test P4**. Similarly to what was observed in other tests, the execution times without triggers remain relatively stable across different database sizes, suggesting that the operation itself scales weakly with the total number of nodes and relationships.

When triggers are enabled, however, execution times increase significantly, with values roughly two to five times higher depending on the database size. This behavior is expected, as the creation of a new `Ride` node activates several triggers that enforce data consistency by performing additional validation steps and updates on related entities.

In particular, each new `Ride` requires the database to check constraints related to its associations with `Driver`, `Passenger`, and possibly `Reservation` nodes. These checks involve matching and filtering existing relationships, operations whose complexity grows with the local connectivity of the graph rather than its overall size.

Therefore, the lack of a clear correlation between total database size and execution time can be explained by the fact that the structure of the relationships being validated remains consistent across datasets: each `Ride` creation triggers a fixed set of operations whose cost depends primarily on the number of linked nodes rather than on the global scale of the data.

5.2.5 P5: Changing the address of a person

Table 8 shows the results for **Test P5**. Again we notice a more or less constant time as it regards the execution without triggers and we don't see a clear correlation between database size and time. Actually for this test we expected to see a correlation seems

	DB13	DB25	DB40	DB70	DB100
Without Triggers	12.598	22.384	13.044	25.553	13.701
With Triggers	42.147	65.271	36.572	37.329	70.449

Table 7: P4 execution times

	DB13	DB25	DB40	DB70	DB100
Without Triggers	11.593	14.082	12.141	13.757	14.680
With Triggers	27.433	26.414	30.700	25.430	31.069

Table 8: P5 execution times

we are eliminating a relation and the validation of the constraint on the relationships from `User` to `HomeAddress` requires the database to fetch all the nodes related to `User`, filtering those that bear the label `HomeAddress` and checking their number. This operation is done twice, first to check that they are more than one, then to check that they are less than two.

A closer inspections reveals that although it is true that the speed of such operations depends on the number of relations to `Reservation` and `Ride` nodes borne by the `User` node, and that the number of such nodes is dependent on the size of the database it is not true that the number of relations from is dependent on the size of the database as the process of database population always loads all the `Reservations` and `Rides` for every `User` that is loaded.

5.2.6 N1: Creating a new ride without defining a starting time

The test correctly returns the error relative to the missing property:
`Constraint successfully triggered: Error executing triggers {ride_startedat_must_be_present=Failed to invoke procedure 'apoc.util.validate': Caused by: java.lang.RuntimeException: startedAt property of Ride must be present}`

The results from the time measurements of **Test N1** can be found on **Table 9**.

5.2.7 N2: Creating a new car without a model

The test correctly returns the error relative to the missing relationship to a car model:
`Constraint successfully triggered: Error executing triggers {car_must_be_in_relationship_with_carmodel_labelassigned=Failed to invoke procedure 'apoc.util.validate': Caused by: java.lang.RuntimeException: Car label must be in relationship with CarModel label}`

The results from the time measurements of **Test N2** can be found on **Table 10**.

DB13	DB25	DB40	DB70	DB100
27.183	30.273	44.138	33.148	38.888

Table 9: N1 execution times

DB13	DB25	DB40	DB70	DB100
24.940	40.979	26.546	28.469	24.841

Table 10: N2 execution times

DB13	DB25	DB40	DB70	DB100
45.668	23.483	26.540	36.727	35.351

Table 11: N3 execution times

5.2.8 N3 and N3alt: Creating a new ride without it being related to a reservation

The tests correctly return the error relative to the missing relationship to a reservation:
`Constraint successfully triggered: Error executing triggers {ride_must_be_in_relationship_with_ongoingreservation_labelassigned= Failed to invoke procedure 'apoc.util.validate': Caused by: java.lang.RuntimeException: Ride label must be in relationship with OngoingReservation label}`

The results from the time measurements of **Test N3** and **Test N3alt** can be found on **Table 11** and **Table 12**. We can see the time taken by **Test N3alt** is consistently better then the time needed for **Test N3**, even though, as it has been better analyzed in the section relative to **Test P2** we know the improvement is small enough to be easily obscured by random fluctuations.

5.2.9 N4: Creating a new reservation without associating a car

The tests correctly return the error relative to the missing relationship to a car:
`Constraint successfully triggered: Error executing triggers {reservation_must_be_in_relationship_with_reservedcar_labelassigned= Failed to invoke procedure 'apoc.util.validate': Caused by: java.lang.RuntimeException: Reservation label must be in relationship with ReservedCar label}`

The results from the time measurements of **Test N4** can be found on **Table 13**.

5.2.10 N5: Creating a new reservation by an unregistered person

The tests correctly return the error relative to the missing `User` label:
`Constraint successfully triggered: Error executing triggers {reserver_must_be_with_user=Failed to invoke procedure 'apoc.util.validate': Caused by: java.lang.RuntimeException: Reserver label must be in a combination with User labels}`

The results from the time measurements of **Test N5** can be found on **Table 14**.

DB13	DB25	DB40	DB70	DB100
25.351	22.787	26.139	26.440	33.377

Table 12: N3alt execution times

DB13	DB25	DB40	DB70	DB100
28.446	25.337	24.463	26.216	31.627

Table 13: N4 execution times

DB13	DB25	DB40	DB70	DB100
30.240	29.760	30.866	30.905	37.792

Table 14: N5 execution times

5.2.11 Duration of the population fo the database

Unlike for the other tests the timings for the population of the database have been calculated a single time, which makes them more vulnerable to fluctuations which must be take into account when analyzing the results. Better results could be done by rerunning the tests multiple times but it hasn't been done for the present thesis due to the excessive time needed for such tests and because conclusions are not expected to differ much from those obtained from the other results.

The results are available in **Table 15**. The growth in time complexity relative to the size of the database grows as expected but we don't see any clear difference in complexity when populating the database with triggers enabled and without, of course part of this is very likely just due to chance for the reasons previously explained - most notably the case for the maximum size where the database without constraints took more time to complete the population then the database with constraints.

	DB13	DB25	DB40	DB70	DB100
Without Triggers	460964.84	885260.73	1438760.48	2656183.83	3870398.74
With Triggers	505980.73	836628.50	1441909.68	3089684.52	2391799.47

Table 15: Times needed for populating the database for each size

SECTION 6
Conclusion

6.1 Summary of Findings

The tests performed across the different scenarios reveal a clear and consistent trend: the presence of triggers significantly impacts query execution times, often increasing them by a factor of three or more compared to scenarios without triggers. This performance penalty is particularly evident in operations that involve multiple node creations and consquently complex validations, as observed in P1 (adding a new User) and P3 (creating a new Car).

Interestingly, while one might expect the database size to correlate with execution time the results show no consistent linear relationship. For most queries (especially those involving only writes or local checks), execution times remain relatively constant across different database sizes. This outcome highlights the efficiency of Neo4j's graph-based model and its native support for constraint checks without costly joins.

While not the main topic of this thesis, as this operations are neither affected nor used by most of the triggers validating the constraints, queries that involve searches or scans before performing write operations (notably P2 and P2alt), execution times do tend to increase with database size. This observation is consistent with the use of node scans and filters (or, to a lesser extent, index seeks) that naturally scale with the number of nodes in the database.

Error-handling tests correctly triggered the intended constraints, confirming the effectiveness of triggers in enforcing data integrity. In addition it has been showed that error-handling doesn't involve degradation of performance compared to successful transactions.

Overall, these findings indicate that while triggers in Neo4j offer effective mechanisms for maintaining data integrity, they introduce a measurable performance overhead. Notably, this overhead remains relatively stable even as the database size grows, suggesting a predictable cost model. When designing systems that leverage triggers, developers must carefully weigh the trade-offs between ensuring strict data consistency and the potential impact on performance. Given their scalability and consistent performance characteristics, triggers can represent a viable solution for applications managing large datasets, particularly where data integrity is a critical priority. However, the decision to use triggers should be informed by the specific workload requirements and tolerance for latency within the application context.

6.2 Limitations and Challenges

A significant limitation encountered during the development of this thesis was the presence of substantial random noise in the measurements. This issue can be at least partially attributed to the lack of system isolation during testing. Specifically, the database was hosted on a personal computer that was simultaneously running other applications, leading to resource contention and unpredictable interference. Further-

more, additional variability was introduced by the operating system's buffering and caching mechanisms, for which no mitigation strategies were implemented. These factors collectively reduced the consistency and reliability of the performance data, complicating the evaluation and comparison of different approaches.

6.3 Further Work

Further work in this research field should prioritize comparing the performance of this approach with similar strategies implemented using relational databases, and of course updating the constraint generator to work with more recent releases of the Neo4j, that is Neo4j 5.

For performance testing, future studies should consider larger databases and more complex schemas and queries to more accurately reflect real-world scenarios. Even better, tests could be designed using the actual distribution of queries that a given application typically executes, providing a more realistic picture of the impact on performance.

Regarding performance optimization, research could focus on integrating this approach with pre-validation mechanisms, which would catch invalid data earlier and reduce unnecessary load on the database. Additionally, further investigation could explore how knowledge extracted from the OntoUML diagram might guide database optimization, for example by identifying attributes known to be unique from the model and leveraging that insight to create indexes, as was suggested in the case of the properties P2 and P2alt in this thesis, as well as other possible indexes that can be identified.

APPENDIX A

Comments from the constraint generator

```
// ----- ATTRIBUTES

PROPERTY_MUST_BE_OF_DATATYPE(Person, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Person, firstName, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Person, lastName, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Person, dateOfBirth, DATE)
PROPERTY_MUST_BE_OF_DATATYPE(Person, gender, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Registration, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Registration, registrationDate, DATE)
PROPERTY_MUST_BE_OF_DATATYPE(Car, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Car, carName, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Car, manufactureYear, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Car, color, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Car, codeVIN, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Car, licencePlate, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Car, locked, BOOLEAN)
PROPERTY_MUST_BE_OF_DATATYPE(Reservation, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Reservation, pricePerKilometer, FLOAT)
PROPERTY_MUST_BE_OF_DATATYPE(Reservation, pricePerHour, FLOAT)
PROPERTY_MUST_BE_OF_DATATYPE(Reservation, freePricing, BOOLEAN)
PROPERTY_MUST_BE_OF_DATATYPE(Reservation, reason, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Ride, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Ride, startedAt, DATE)
PROPERTY_MUST_BE_OF_DATATYPE(Ride, finishedAt, DATE)
PROPERTY_MUST_BE_OF_DATATYPE(Ride, initialMileage, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Ride, finalMileage, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Ride, initialFuelLevel, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Ride, finalFuelLevel, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(HomeAddress, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(HomeAddress, street, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(HomeAddress, city, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(HomeAddress, country, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(CarModel, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(CarModel, modelName, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(CarModel, numberOfSeats, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(CarModel, tankCapacity, FLOAT)
PROPERTY_MUST_BE_OF_DATATYPE(CarModel, trunkCapacity, FLOAT)
```

```
PROPERTY_MUST_BE_OF_DATATYPE(Account, id, INTEGER)
PROPERTY_MUST_BE_OF_DATATYPE(Account, password, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Account, language, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Account, note, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(Account, promoCode, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(User, userEmail, STRING)
PROPERTY_MUST_BE_OF_DATATYPE(User, userPhone, STRING)
PROPERTY_MUST_BE_PRESENT(Person, id)
PROPERTY_MUST_BE_PRESENT(Person, firstName)
PROPERTY_MUST_BE_PRESENT(Person, lastName)
PROPERTY_MUST_BE_PRESENT(Person, dateOfBirth)
PROPERTY_MUST_BE_PRESENT(Person, gender)
PROPERTY_MUST_BE_PRESENT(Registration, id)
PROPERTY_MUST_BE_PRESENT(Registration, registrationDate)
PROPERTY_MUST_BE_PRESENT(Car, id)
PROPERTY_MUST_BE_PRESENT(Car, carName)
PROPERTY_MUST_BE_PRESENT(Car, manufactureYear)
PROPERTY_MUST_BE_PRESENT(Car, color)
PROPERTY_MUST_BE_PRESENT(Car, codeVIN)
PROPERTY_MUST_BE_PRESENT(Car, licencePlate)
PROPERTY_MUST_BE_PRESENT(Car, locked)
PROPERTY_MUST_BE_PRESENT(Reservation, id)
PROPERTY_MUST_BE_PRESENT(Reservation, pricePerKilometer)
PROPERTY_MUST_BE_PRESENT(Reservation, pricePerHour)
PROPERTY_MUST_BE_PRESENT(Reservation, freePricing)
PROPERTY_MUST_BE_PRESENT(Reservation, reason)
PROPERTY_MUST_BE_PRESENT(Ride, id)
PROPERTY_MUST_BE_PRESENT(Ride, startedAt)
PROPERTY_MUST_BE_PRESENT(Ride, finishedAt)
PROPERTY_MUST_BE_PRESENT(Ride, initialMileage)
PROPERTY_MUST_BE_PRESENT(Ride, finalMileage)
PROPERTY_MUST_BE_PRESENT(Ride, initialFuelLevel)
PROPERTY_MUST_BE_PRESENT(Ride, finalFuelLevel)
PROPERTY_MUST_BE_PRESENT(HomeAddress, id)
PROPERTY_MUST_BE_PRESENT(HomeAddress, street)
PROPERTY_MUST_BE_PRESENT(HomeAddress, city)
PROPERTY_MUST_BE_PRESENT(HomeAddress, country)
PROPERTY_MUST_BE_PRESENT(CarModel, id)
PROPERTY_MUST_BE_PRESENT(CarModel, modelName)
PROPERTY_MUST_BE_PRESENT(CarModel, numberOfSeats)
PROPERTY_MUST_BE_PRESENT(CarModel, tankCapacity)
PROPERTY_MUST_BE_PRESENT(CarModel, trunkCapacity)
PROPERTY_MUST_BE_PRESENT(Account, id)
PROPERTY_MUST_BE_PRESENT(Account, password)
PROPERTY_MUST_BE_PRESENT(Account, language)
PROPERTY_MUST_BE_PRESENT(Account, note)
PROPERTY_MUST_BE_PRESENT(Account, promoCode)
PROPERTY_MUST_BE_PRESENT(User, userEmail)
```

```
PROPERTY_MUST_BE_PRESENT(User, userPhone)
PROPERTY_MUST_BE_UNIQUE(Person, id)
PROPERTY_MUST_BE_UNIQUE(Registration, id)
PROPERTY_MUST_BE_UNIQUE(Car, id)
PROPERTY_MUST_BE_UNIQUE(Reservation, id)
PROPERTY_MUST_BE_UNIQUE(Ride, id)
PROPERTY_MUST_BE_UNIQUE(HomeAddress, id)
PROPERTY_MUST_BE_UNIQUE(CarModel, id)
PROPERTY_MUST_BE_UNIQUE(Account, id)

// ----- STEREOTYPES

LABEL_CANNOT_BE_IN_COMBINATION(Person,
    {Registration, Car, Reservation, Ride, HomeAddress, Account})
LABEL_CANNOT_BE_IN_COMBINATION(Registration,
    {Person, Car, Reservation, Ride, HomeAddress, Account})
LABEL_CANNOT_BE_IN_COMBINATION(Car,
    {Person, Registration, Reservation, Ride, HomeAddress, Account})
LABEL_CANNOT_BE_IN_COMBINATION(Reservation,
    {Person, Registration, Car, Ride, HomeAddress, Account})
LABEL_CANNOT_BE_IN_COMBINATION(Ride,
    {Person, Registration, Car, Reservation, HomeAddress, Account})
LABEL_CANNOT_BE_IN_COMBINATION(HomeAddress,
    {Person, Registration, Car, Reservation, Ride, Account})
LABEL_CANNOT_BE_IN_COMBINATION(Account,
    {Person, Registration, Car, Reservation, Ride, HomeAddress})
LABEL_CANNOT_BE_REMOVED(Person)
LABEL_CANNOT_BE_REMOVED(Registration)
LABEL_CANNOT_BE_REMOVED(Car)
LABEL_CANNOT_BE_REMOVED(Reservation)
LABEL_CANNOT_BE_REMOVED(Ride)
LABEL_CANNOT_BE_REMOVED(HomeAddress)
LABEL_CANNOT_BE_REMOVED(CarModel)
LABEL_CANNOT_BE_REMOVED(Account)

// ----- GENERALIZATIONS

LABEL_MUST_BE_IN_COMBINATION(Reserver, {User})
LABEL_MUST_BE_IN_COMBINATION(Driver, {Reserver})
LABEL_MUST_BE_IN_COMBINATION(DrivenCar, {ReservedCar})
LABEL_MUST_BE_IN_COMBINATION(ReservedCar, {Car})
LABEL_MUST_BE_IN_COMBINATION(User, {Person})
LABEL_MUST_BE_IN_COMBINATION(OngoingReservation, {Reservation})

// ----- GENERALIZATION SETS


// ----- ASSOCIATIONS
```

```
RELATIONSHIP_MUST_BE_PRESENT(User, Registration)
RELATIONSHIP_MUST_BE_PRESENT(Registration, User)
RELATIONSHIP_MUST_BE_PRESENT(Driver, Ride)
RELATIONSHIP_MUST_BE_PRESENT(Ride, Driver)
RELATIONSHIP_MUST_BE_PRESENT(DrivenCar, Ride)
RELATIONSHIP_MUST_BE_PRESENT(Ride, DrivenCar)
RELATIONSHIP_MUST_BE_PRESENT(OngoingReservation, Ride)
RELATIONSHIP_MUST_BE_PRESENT(Ride, OngoingReservation)
RELATIONSHIP_MUST_BE_PRESENT(Person, HomeAddress)
RELATIONSHIP_MUST_BE_PRESENT(Registration, Account)
RELATIONSHIP_MUST_BE_PRESENT(Reserver, Reservation)
RELATIONSHIP_MUST_BE_PRESENT(Reservation, Reserver)
RELATIONSHIP_MUST_BE_PRESENT(ReservedCar, Reservation)
RELATIONSHIP_MUST_BE_PRESENT(Reservation, ReservedCar)
RELATIONSHIP_MUST_BE_PRESENT(Car, CarModel)
RELATIONSHIP_MUST_BE_LIMITED(User, Registration, 1)
RELATIONSHIP_MUST_BE_LIMITED(Registration, User, 1)
RELATIONSHIP_MUST_BE_LIMITED(Ride, Driver, 1)
RELATIONSHIP_MUST_BE_LIMITED(Ride, DrivenCar, 1)
RELATIONSHIP_MUST_BE_LIMITED(Ride, OngoingReservation, 1)
RELATIONSHIP_MUST_BE_LIMITED(Person, HomeAddress, 2)
RELATIONSHIP_MUST_BE_LIMITED(Account, Registration, 1)
RELATIONSHIP_MUST_BE_LIMITED(Registration, Account, 1)
RELATIONSHIP_MUST_BE_LIMITED(Reservation, Reserver, 1)
RELATIONSHIP_MUST_BE_LIMITED(Reservation, ReservedCar, 1)
RELATIONSHIP_MUST_BE_LIMITED(Car, CarModel, 1)
```

APPENDIX B

Tables' sizes

table	#tuples
users	8270
reservations	79922
cars	57
userProfiles	8270
carModels	7
carBrands	1
countries	237
rides	67979
addresses	24894

APPENDIX C

Database Sizes

DB100

label	#nodes
Account	8270
HomeAddress	16535
Car	57
CarBrand	1
CarModel	7
Person	8270
Reservation	79915
Ride	67979

DB70

label	#nodes
Account	4195
HomeAddress	8388
Car	57
CarBrand	1
CarModel	7
Person	4195
Reservation	59646
Ride	50764

DB40

label	#nodes
Account	2066
HomeAddress	4133
Car	57
CarBrand	1
CarModel	7
Person	2066
Reservation	34568
Ride	28840

DB25

label	#nodes
Account	1074
HomeAddress	2147
Car	57
CarBrand	1
CarModel	7
Person	1074
Reservation	22869
Ride	18612

DB13

label	#nodes
Account	530
HomeAddress	1059
Car	57
CarBrand	1
CarModel	7
Person	530
Reservation	11727
Ride	8875

APPENDIX D

Cypher script for the population of the Neo4j Database

```
LOAD CSV WITH HEADERS FROM "file:///csv/cars.csv" AS row
WITH row, CASE row.locked
    WHEN "t" THEN true
    WHEN "f" THEN false
    ELSE false
    END AS locked
CREATE (n:Car)
SET n.id = toInteger(row.id),
n.carName = row.name,
n.manufactureYear = toInteger(row.manufacture_year),
n.color = row.color,
n.codeVIN = row.vin_code,
n.licencePlate = row.licence_number,
n.locked = toBoolean(locked)
WITH n, row
MERGE (cm:CarModel {id: toInteger(row.model_id)})
CREATE (n)-[r:IS_OF_MODEL]->(cm)
SET cm.modelName = "name",
cm.numberOfSeats = 0,
cm.trunkCapacity = toFloat(0),
cm.tankCapacity = toFloat(0);

LOAD CSV WITH HEADERS FROM "file:///csv/carModels.csv" AS row
MATCH (cm:CarModel {id:toInteger(row.id)})
SET cm.modelName = row.name,
cm.numberOfSeats = toInteger(row.seats),
cm.trunkCapacity = toFloat(row.trunk_capacity),
cm.tankCapacity = toFloat(row.tank_capacity);

LOAD CSV WITH HEADERS FROM "file:///csv/userProfiles.csv" AS row
CREATE (p:Person {id: toInteger(row.user_id)})
SET p.firstName = row.first_name,
```

```
p.lastName = row.last_name,
p.dateOfBirth = date(replace(row.date_of_birth, " ", "T")),
p.gender = row.gender,
p.userPhone = row.phone
WITH p, row
MERGE (a3:HomeAddress {id: toInteger(row.home_address_id)})
SET a3.city = "city",
a3.country = "country",
a3.street = "street"
WITH a3, p, row
CREATE (p)-[:LIVES_IN]->(a3)
WITH p, row
MERGE (a4:HomeAddress {id:
 ↪  toInteger(row.correspondence_address_id)})
SET a4.city = "city",
a4.country = "country",
a4.street = "street"
WITH a4, p, row
CREATE (p)-[:HAS_CORRESPONDANCE_ADDRESS]->(a4)
WITH row
MERGE (a: Account {id: toInteger(row.id)})
SET a.language = row.language,
a.user_id = toInteger(row.user_id),
a.note = "nn",
a.password = "nn",
a.promoCode = "nn";

LOAD CSV WITH HEADERS FROM "file:///csv/users.csv" AS row
WITH row
MATCH (p: Person {id: toInteger(row.id)})
SET p.userEmail = row.email_lc
with row, p,
    CASE WHEN  row.promo_code IS NULL OR row.promo_code = ""
        THEN "null" ELSE row.promo_code END AS promoCode,
    CASE WHEN  row.note IS NULL OR row.note = ""
        THEN "null" ELSE row.note END AS note
MATCH (a:Account {user_id: toInteger(row.id)})
SET a.password = row.password,
a.note = note,
a.promoCode = promoCode
WITH a, p, row
CREATE (p) -[:REGISTERED]-> (r:Registration {id: toInteger(row.id),
```

```
    registrationDate: date(datetime(replace(row.added_at, " ",
    ↪ "T")))})
CREATE (r)-[:HAS_ACCOUNT]->(a)
SET p:User;

USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///csv/reservations.csv" AS row
WITH row,
CASE row.free_pricing WHEN "t" THEN true WHEN "f"
    THEN false ELSE false END AS freePricing,
CASE WHEN row.reason IS NULL OR row.reason = ""
    THEN "none" ELSE row.reason END AS reasonClean,
CASE WHEN row.price_per_km IS NULL OR row.price_per_km = ""
    THEN toFloat(0) ELSE toFloat(row.price_per_km) END AS
    ↪ pricePerKilometer,
CASE WHEN row.price_per_hour IS NULL OR row.price_per_hour = ""
    THEN toFloat(0) ELSE toFloat(row.price_per_hour) END AS
    ↪ pricePerHour
MATCH (u:User {id: toInteger(row.user_id)}), (c:Car {id:
↪ toInteger(row.car_id)})
CREATE (r:Reservation_Importing  {id: toInteger(row.id),
    pricePerKilometer : pricePerKilometer, pricePerHour :
    ↪ pricePerHour,
    freePricing : freePricing, reason: reasonClean})
CREATE (u)-[:RESERVED]->(r)
CREATE (c)-[:IS_RESERVED]->(r);

MATCH (u:User)-[:RESERVED]-> (r:Reservation_Importing)
↪ <-[:IS_RESERVED]-(c:Car)
REMOVE r:Reservation_Importing
SET r:Reservation
SET u:Reserver
SET c:ReservedCar;

USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///csv/rides.csv" AS row
WITH row,
date(datetime(replace(row.started_at, " ", "T"))) as startedAt,
CASE WHEN row.finished_at IS NULL OR row.finished_at = ""
    THEN date(datetime(replace(row.started_at, " ", "T")))
    ELSE date(datetime(replace(row.finished_at, " ", "T"))) END AS
    ↪ finishedAt
```

```
MATCH (res: Reservation {id: toInteger(row.reservation_id)}),
    (u:User)-[:RESERVED]->(res), (c:Car)-[:IS_RESERVED]->(res)
CREATE (rr:Ride_Importing  {id: toInteger(row.id), startedAt :
 ↪  startedAt,
    finishedAt : finishedAt, initialMileage :
     ↪  toInteger(row.initial_mileage),
    finalMileage: toInteger(row.final_mileage),
    initialFuelLevel: toInteger(row.initial_tank_capacity),
    finalFuelLevel: toInteger(row.final_tank_capacity)})
CREATE (rr)-[:FROM_RESERVATION]->(res)
CREATE (u)-[:RIDES]->(rr)
CREATE (c)-[:IS_DRIVEN]->(rr);

MATCH (u:User)-[:RIDES]->(r:Ride_Importing)<-[:IS_DRIVEN]-(c:Car),
    (r)-[:FROM_RESERVATION]->(res)
REMOVE r:Ride_Importing
SET r:Ride,
res:OngoingReservation,
u:Driver,
c:DrivenCar;

LOAD CSV WITH HEADERS FROM "file:///csv/addresses.csv" AS row
MATCH (aa:HomeAddress {id: toInteger(row.id)})
SET aa.country = row.country_id,
aa.city = row.city,
aa.street = row.street;
LOAD CSV WITH HEADERS FROM "file:///csv/countries.csv" AS row
MATCH (a:HomeAddress)
WHERE a.country = row.id
SET a.country = row.name;
```

Listing 23: Cypher script for the population of he Neo4j Database

APPENDIX E

Summary of tests

Name	Description	Involved Classes
P1	Adding new User	Person User Account Registration HomeAddress
P2	Adding a new reservation	User Reserver Car ReservedCar Reservation
P2alt	Alternative to P2 where User is substituted by Person to leverage the index generated by the unique property constraint	Person Reserver Car ReservedCar Reservation
P3	Creating a new car	Car CarModel
P4	Creating a new ride	Reservation OngoingReservation Ride User Driver Car DrivenCar
P5	Changing the address of a person	Person HomeAddress

Table 16: Summary of positive tests

Name	Description	Involved Classes
N1	Creating a new ride without defining a starting time	Reservation OngoingReservation Ride User Reserver Driver Car ReservedCar DrivenCar
N2	Creating a new car without a model	Car
N3	Creating a new ride without it being related to a reservation	User Car Ride Driver DrivenCar
N3alt	Creating a new ride without it being related to a reservation, leveraging the index on Person in the match	Person Car Ride Driver DrivenCar
N4	Creating a new reservation without associating a car	User Reserver Reservation
N5	Creating a new reservation by an unregistered person	Car ReservedCar Person Reserver HomeAddress Reservation

Table 17: Summary of negative tests

Name	Constraint
N1	PROPERTY_MUST_BE_PRESENT(Ride, startedAt)
N2	RELATIONSHIP_MUST_BE_PRESENT(Car, CarModel)
N3	RELATIONSHIP_MUST_BE_PRESENT(Ride, OngoingReservation)
N3alt	RELATIONSHIP_MUST_BE_PRESENT(Ride, OngoingReservation)
N4	RELATIONSHIP_MUST_BE_PRESENT(Reservation, Car)
N5	LABEL_MUST_BE_IN_COMBINATION(Reserver, User)

Table 18: Negative tests and violated constraints

APPENDIX F

Hardware and Software specifications

Component	Specification
CPU	11th Gen Intel(R) Core(TM) i5-1135G, 4 cores / 8 threads, 2.40GHz
RAM	8,00 GB, 3200 MHz
Storage	512 GB NVMe SSD (NVMe INTEL SSDPEKNW512GZL)
Operating System	Windows 11 Home
Java Version	OpenJDK 11.0.26.4
Neo4j Version	Neo4j 4.4.42
Python Version	Python 3.10.11
Neo4j Driver	neo4j-python-driver 5.28

Table 19: Hardware and software configuration used for benchmarking.

APPENDIX G

Code for the JSONImporter

```
using Neo4jConstraintGenerator.OntoUML;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Security.Principal;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;

namespace Neo4jConstraintGenerator
{
    internal class JSONImporter
    {
        string filePath;
        JsonElement JSONmodel;
        public JSONImporter(string filePath) {
            this.filePath = filePath;
        }

        public Model CreateModelFromJSON()
        {
            string json = File.ReadAllText(filePath);
            using JsonDocument doc = JsonDocument.Parse(json);
            JsonElement root = doc.RootElement;

            // data stuctures containing the processed element that
            ↪ make up the internal model
            Dictionary<string, Class> classes = new
            ↪ Dictionary<string, Class>();
            Dictionary<string, DataType> types = new
            ↪ Dictionary<string, DataType>();
            Dictionary<String, Generalization> generalizations =
                new Dictionary<String, Generalization>();
            List<Association> associations = new
            ↪ List<Association>();
```

```
33              List<GeneralizationSet> generalizationSets = new
                ↪  List<GeneralizationSet>();
34
35              //Lists organizing the different elements in categories
                ↪  to allow ordered processing
36              List<JsonElement> JSONdatatypes = new
                ↪  List<JsonElement>();
37              List<JsonElement> JSONclasses = new List<JsonElement>();
38              List<JsonElement> JSONgeneralizations = new
                ↪  List<JsonElement>();
39              List<JsonElement> JSONrelations = new
                ↪  List<JsonElement>();
40              List<JsonElement> JSONgeneralizationSets = new
                ↪  List<JsonElement>();
41
42
43
44              JSONmodel =
                ↪  root.GetProperty("model").GetProperty("contents");
45
46              //cycling through the JSON elements and categorizing
                ↪  them
47              foreach (JsonElement element in
                ↪  JSONmodel.EnumerateArray())
48              {
49                  string type =
                    ↪  element.GetProperty("type").ToString();
50                  switch (type)
51                  {
52                      case "Class":
53                          if (!element.GetProperty("type").GetString()
54                              .Equals("Class")) continue;
55                          string stereotypeString =
                            ↪  element.GetProperty("stereotype")
56                              .GetString();
57                          string id =
                            ↪  element.GetProperty("id").GetString();
58                          if (stereotypeString.Equals("datatype"))
                            ↪  JSONdatatypes.Add(element);
59                          else JSONclasses.Add(element);
60                          break;
61                      case "Generalization":
```

```
62                          JSONgeneralizations.Add(element);
63                          break;
64                      case "Relation":
65                          JSONrelations.Add(element);
66                          break;
67                      case "GeneralizationSet":
68                          JSONgeneralizationSets.Add(element);
69                          break;
70                      default:
71                          break;
72                  }
73              }
74
75              //processing elements from each category
76              foreach (JsonElement element in JSONdatatypes)
77              {
78                  string id = element.GetProperty("id").GetString();
79                  string typestring =
                    ↪  element.GetProperty("name").GetString();
80                  switch (typestring)
81                  {
82                      case "int":
83                      case "integer":
84                          types.Add(id, DataType.Integer);
85                          break;
86                      case "string":
87                          types.Add(id, DataType.String);
88                          break;
89                      case "float":
90                          types.Add(id, DataType.Float);
91                          break;
92                      case "datetime":
93                          types.Add(id, DataType.Date);
94                          break;
95                      case "boolean":
96                          types.Add(id, DataType.Boolean);
97                          break;
98                      default:
99                          types.Add(id, new DataType(typestring));
100                         break;
101                 }
102             }
```

```
103             foreach (JsonElement element in JSONclasses){
104                 string stereotypeString =
                    ↪ element.GetProperty("stereotype").ToString();
105                 if (stereotypeString.Equals("datatype")) continue;
106                 ClassStereotype stereotype =
                    ↪ lookupStereotype(stereotypeString);
107                 string className =
                    ↪ element.GetProperty("name").GetString();
108                 Class newClass = new Class(className, stereotype);
109                 Debug.WriteLine("class " + className);
110                 JsonElement properties =
                    ↪ element.GetProperty("properties");
111                 if(properties.ValueKind != JsonValueKind.Null)
112                     foreach (JsonElement property in
                        ↪ properties.EnumerateArray())
113                 {
114                     JsonElement propertyType =
                        ↪ property.GetProperty("propertyType")
115                         .GetProperty("id");
116                     DataType dataType = propertyType.ValueKind ==
                        ↪ JsonValueKind.Null ?
117                     DataType.String :
                        ↪ types[propertyType.ToString()];
118                     string attributeName =
                        ↪ property.GetProperty("name").GetString();
119                     bool isID = attributeName.Equals("id");
120                     newClass.Attributes.Add(new
                        ↪ ClassAttribute(attributeName, dataType,
                        ↪ isID));
121                     Debug.WriteLine("Attribute " + attributeName
122                         + " of type " + dataType.ToString());
123                 }
124                 classes.Add(element.GetProperty("id").GetString(),
                    ↪ newClass);
125             }
126             foreach (JsonElement element in JSONgeneralizations)
127             {
128                 string idGen = element.GetProperty("general")
129                     .GetProperty("id").GetString();
130                 string idSpec = element.GetProperty("specific")
131                     .GetProperty("id").GetString();
132
```

```
133                 generalizations.Add(element
134                         .GetProperty("id").GetString(),
135                     new Generalization(classes[idGen],
                        ↪ classes[idSpec]));
136                 Debug.WriteLine("Generalization of " +
137                 classes[idSpec].Name + " to " +
                    ↪ classes[idGen].Name);
138             }
139             foreach (JsonElement element in JSONrelations)
140             {
141                 Class class1, class2;
142                 Multiplicity multiplicity1, multiplicity2;
143                 string id1 = element.GetProperty("properties")[0]
144                     .GetProperty("propertyType").GetProperty("id")
145                     .GetString();
146                 string id2 = element.GetProperty("properties")[1]
147                     .GetProperty("propertyType").GetProperty("id")
148                     .GetString();
149                 class1 = classes[id1];
150                 class2 = classes[id2];
151                 multiplicity1 = lookupMultiplicity(element
152                     .GetProperty("properties")[0]
153                     .GetProperty("cardinality").GetString());
154                 multiplicity2 = lookupMultiplicity(element
155                     .GetProperty("properties")[1]
156                     .GetProperty("cardinality").GetString());
157                 associations.Add(new Association(class1, class2,
                    ↪ multiplicity1, multiplicity2));
158                 Debug.WriteLine("Association of classes " +
                    ↪ class1.Name + "-" + class2.Name +
159                     " with multiplicity (" + multiplicity1 + "," +
                        ↪ multiplicity2 + ")");
160             }
161             foreach (JsonElement element in JSONgeneralizationSets)
162             {
163                 bool isDisjoint, isComplete;
164                 isDisjoint =
                    ↪ element.GetProperty("isDisjoint").GetBoolean();
165                 isComplete =
                    ↪ element.GetProperty("isComplete").GetBoolean();
166                 List<Generalization> list = new
                    ↪ List<Generalization>();
```

```
                foreach (JsonElement generalization in
                    element.GetProperty("generalizations")
                    .EnumerateArray())
                {
                    list.Add(generalizations[generalization
                        .GetProperty("id").GetString()]);
                }
                generalizationSets.Add(new GeneralizationSet(list,
                ↪  isComplete, isDisjoint));
            }


            //Constructing the internal model
            Model model = new Model()
            {
                Classes = classes.Values.ToList(),
                Generalizations = generalizations.Values.ToList(),
                Associations = associations,
                GeneralizationSets = generalizationSets
            };

            return model;
        }

        ClassStereotype lookupStereotype(string stereotypeString)
        {
            ClassStereotype stereotype = ClassStereotype.None;
            switch (stereotypeString)
            {
                case "kind":
                    stereotype = ClassStereotype.Kind;
                    break;
                case "relator":
                    stereotype = ClassStereotype.Relator;
                    break;
                case "mode":
                    stereotype = ClassStereotype.Mode;
                    break;
                case "quality":
                    stereotype = ClassStereotype.Quality;
                    break;
                case "role":
```

```
208                     stereotype = ClassStereotype.Role;
209                     break;
210                 case "subkind":
211                     stereotype = ClassStereotype.Subkind;
212                     break;
213                 case "phase":
214                     stereotype = ClassStereotype.Phase;
215                     break;
216                 case "mixin":
217                     stereotype = ClassStereotype.PhaseMixin;
218                     break;
219                 case "roleMixin":
220                     stereotype = ClassStereotype.PhaseMixin;
221                     break;
222                 case "phaseMixin":
223                     stereotype = ClassStereotype.PhaseMixin;
224                     break;
225                 case "event":
226                     stereotype = ClassStereotype.PhaseMixin;
227                     break;
228                 case "historicalRole":
229                     stereotype = ClassStereotype.HistoricalRole;
230                     break;
231                 case "type":
232                     stereotype= ClassStereotype.Type;
233                     break;
234                 case "datatype":
235                     break;
236                 default:
237                     break;
238             }
239             return stereotype;
240         }
241
242         Multiplicity lookupMultiplicity(string s)
243         {
244             switch (s)
245             {
246                 case "1":
247                     return Multiplicity.ExactlyOne;
248                 case "*":
249                 case "0..*":
```

```
250                     return Multiplicity.ZeroToMany;
251                 case "1..*":
252                     return Multiplicity.OneToMany;
253                 case "0..1":
254                     return Multiplicity.ZeroToOne;
255                 default:
256                     string[] parts = s.Split("..");
257
258                     if (parts.Length == 2 && int.TryParse(parts[0],
                        ↪ out int num1)
259                         && int.TryParse(parts[1], out int num2))
260                     {
261                         return new Multiplicity(num1, num2);
262                     }
263                     else
264                     {
265                         throw new NotImplementedException();
266                     }
267
268             }
269         }
270     }
271
272 }
273
```

Listing 24: Code for the JSONImporter

APPENDIX H

Execution plans for the benchmark queries

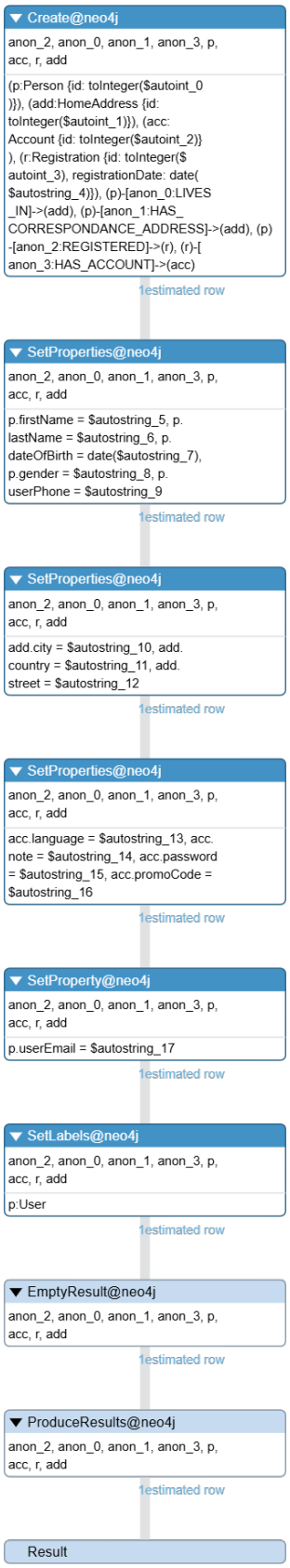

Figure 15: Execution plan of test P1

Figure 16: Execution plan of test P2

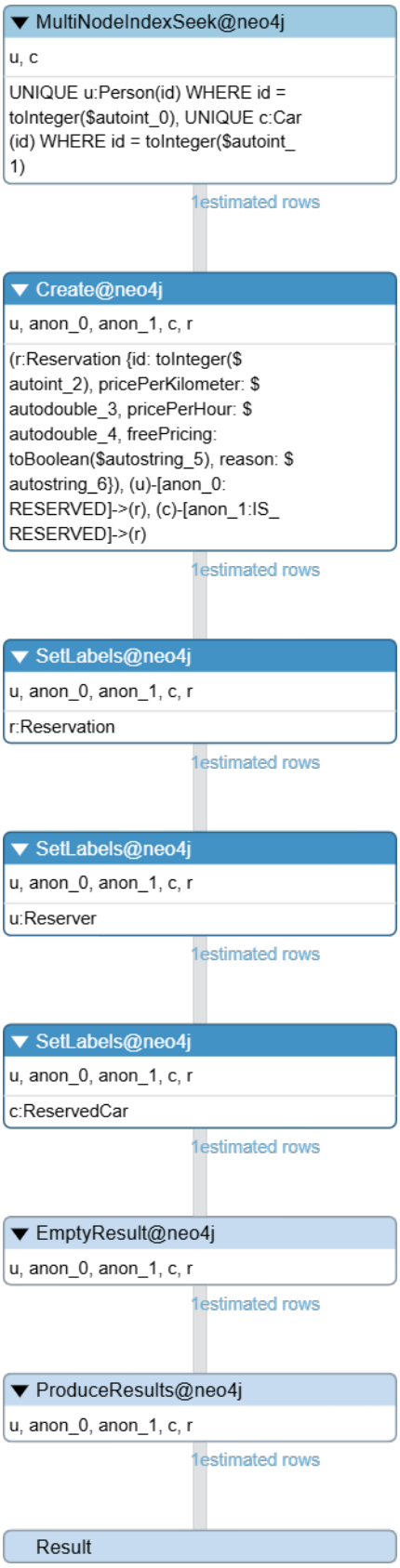

Figure 17: Execution plan of test P2Alt

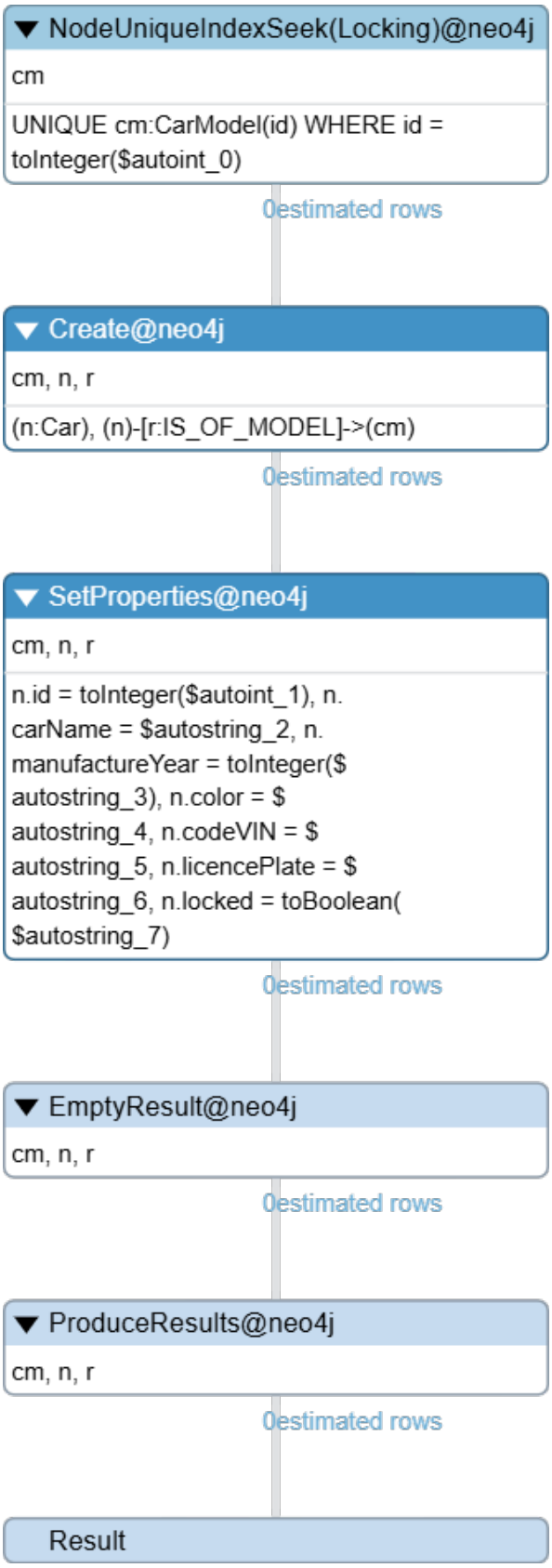

Figure 18: Execution plan of test P3

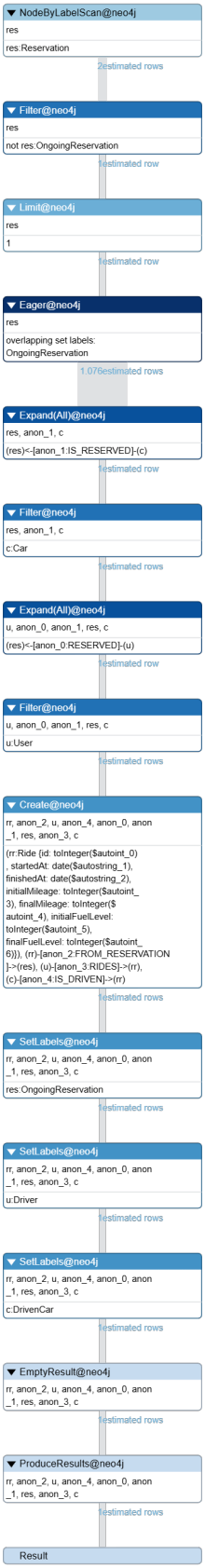

Figure 19: Execution plan of test P4

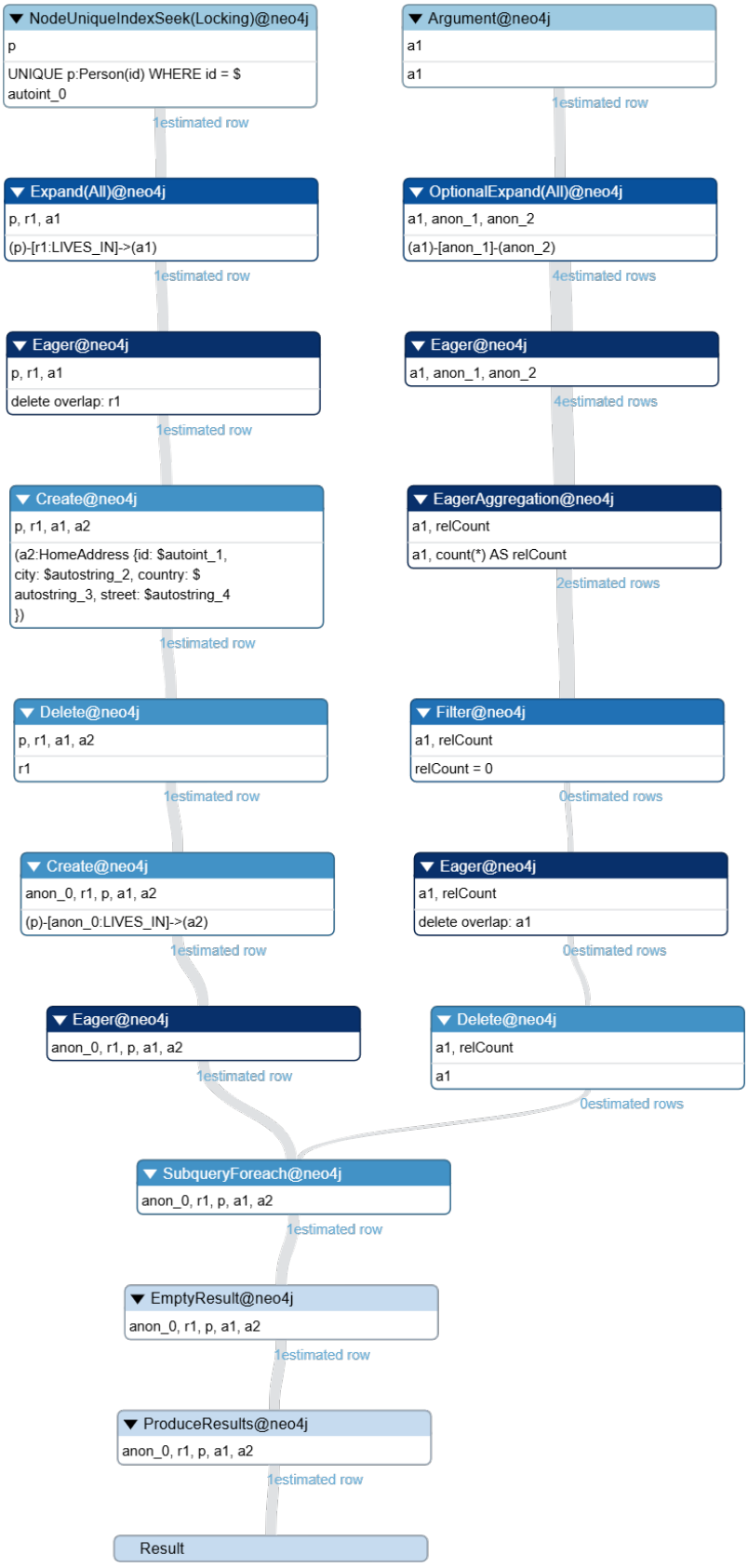

Figure 20: Execution plan of test P5

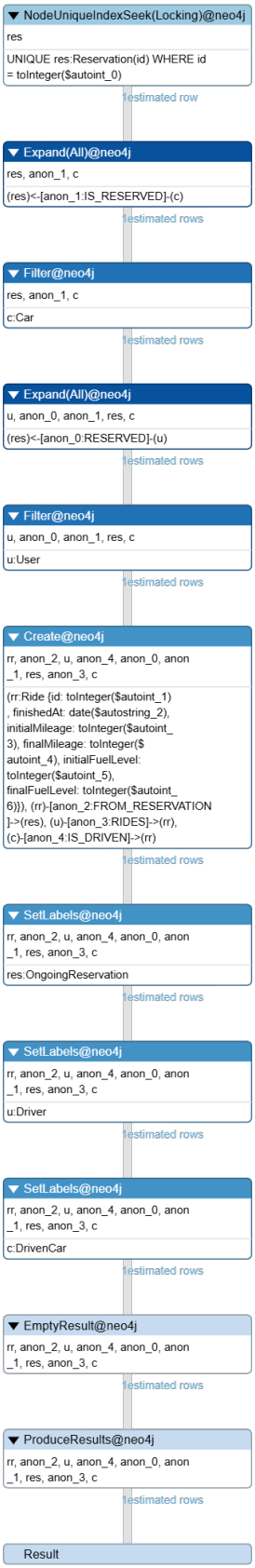

Figure 21: Execution plan of test N1

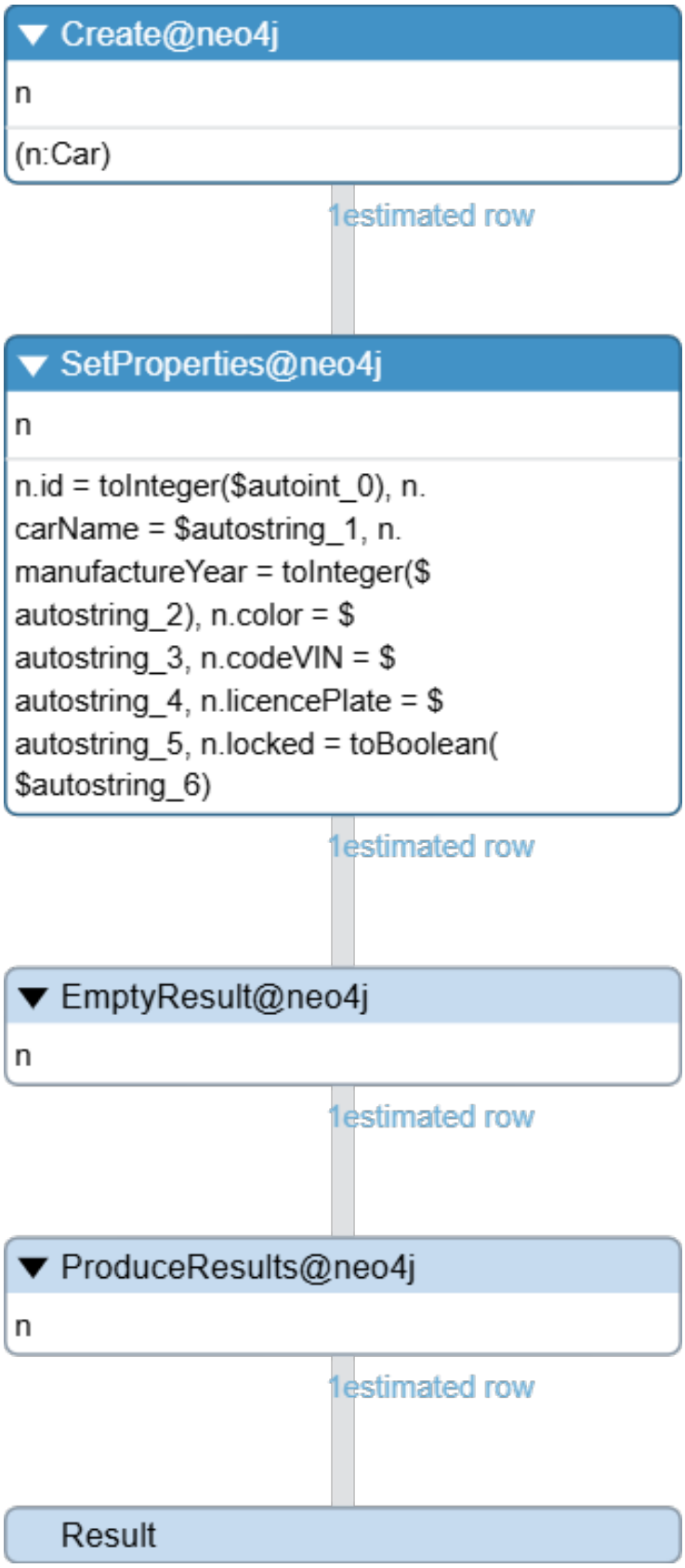

Figure 22: Execution plan of test N2

Figure 23: Execution plan of test N3

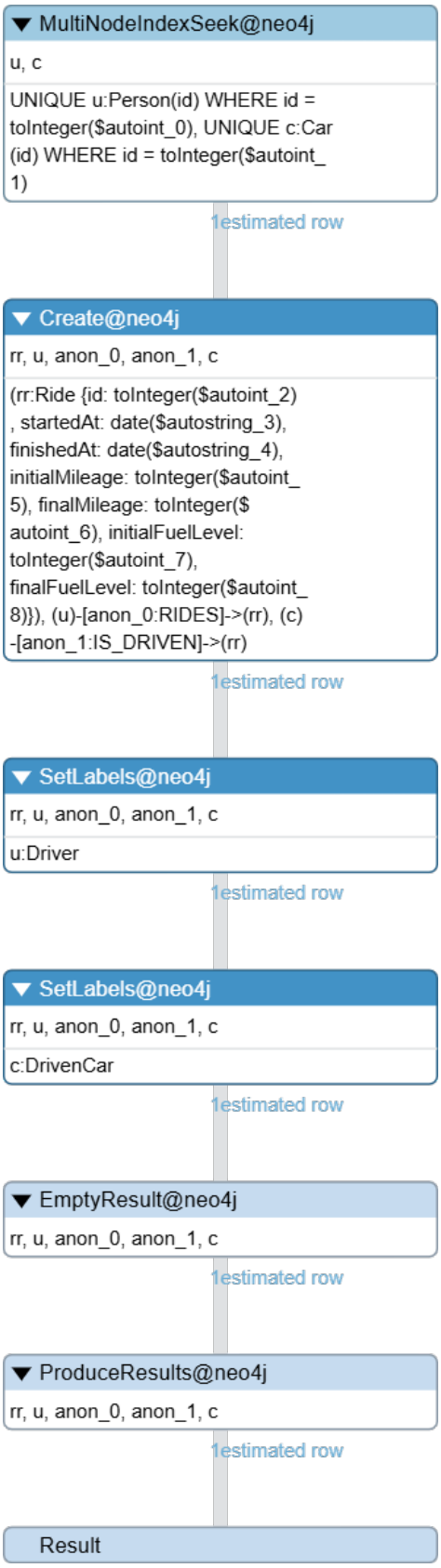

Figure 24: Execution plan of test N3alt

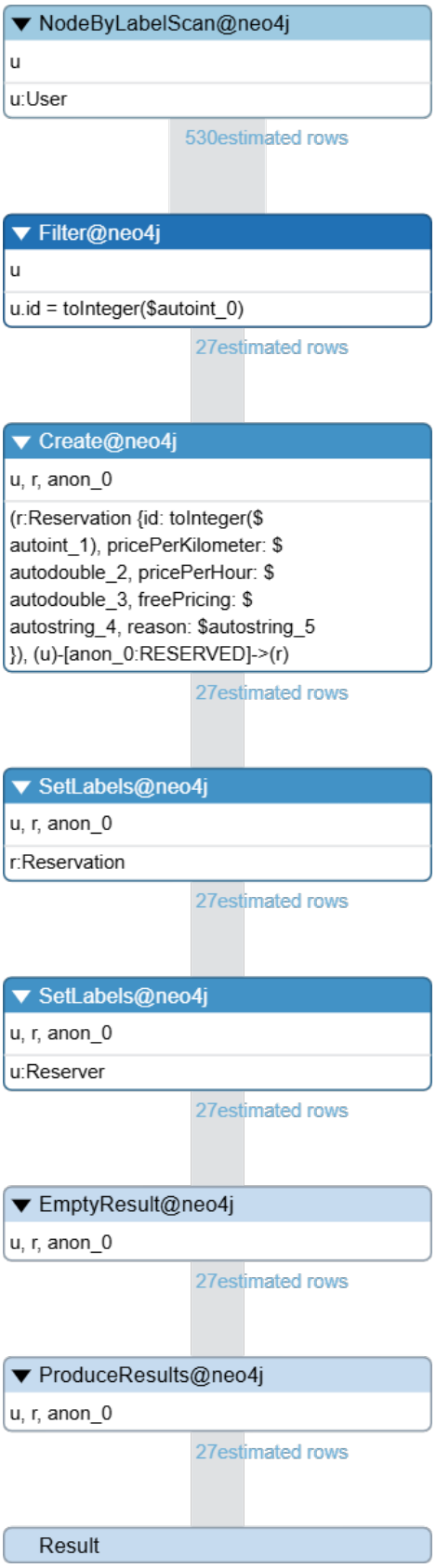

Figure 25: Execution plan of test N4

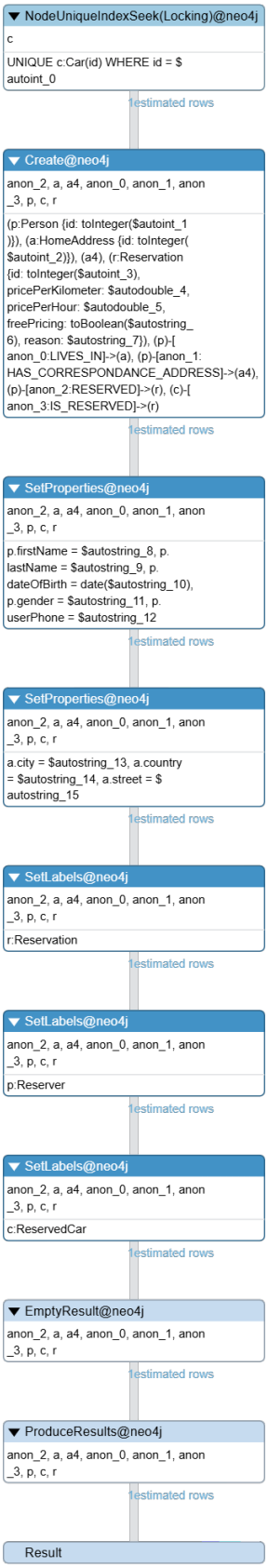

Figure 26: Execution plan of test N5

References

[AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 2008.

[Apo] `https://neo4j.com/docs/apoc/current/overview/apoc/`. Apoc documentation.

[Bar] Pedro Paulo Favato Barcelos. Ontouml json2graph decoder.

[Cyp] Cypher manual at development.neo4j.dev.

[ea10] Bernardo F. B. Braga et al. Transforming ontouml into alloy: Towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering*, 2010.

[ea23] Renata Guizzardi et al. An ontology-based approach to engineering ethicality requirements. *Software and System Modeling*, 2023.

[FSGA19] Camila Ramos Fonseca, Tiago Prince Sales, Giancarlo Guizzardi, and João Paulo A. Almeida. Relations in ontology-driven conceptual modeling. *Lecture Notes in Computer Science*, 2019.

[G+18] Giancarlo Guizzardi et al. Endurant types in ontology-driven conceptual modeling: Towards ontouml 2.0. *Conceptual Modeling - 37th International Conference*, 2018.

[GG+22] Giancarlo Guizzardi, Giancarlo Guarino, et al. Ufo: Unified foundational ontology. *Applied ontology*, 2022.

[Gri24] Guus Grievink. Ontology-driven software development: Generating java code from ontouml. Master's thesis, University of Twente, 2024.

[GSAG21] Giancarlo Guizzardi, Tiago Prince Sales, João Paulo A. Almeida, and Renata S. S. Guizzardi. Types and taxonomic structures in conceptual modeling: A novel ontological theory and engineering support. *Data & Knowledge Engineering*, 2021.

[Gui05] Giancarlo Guizzardi. *Ontological Foundations for Structural Conceptual Models*. PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2005.

[Neo] Neo4j Documentation, https://neo4j.com/docs/.

[Onta] Prince Sales, T. (Creator), M. Fonseca, C. (Creator), Favato Barcelos, P. P. (Creator) (31 Jul 2023). OntoUML Vocabulary. Zenodo. 10.5281/zenodo.8199343.

[Ontb] OntoUML plugin for Visual Paradigm, https://github.com/OntoUML/ontouml-vp-plugin.

[PRV+24] Jaroslav Pokorný, Zdeněk Rybola, Michal Valenta, Jiří Zikán, and Robert Pergl. Instantiation of ontouml models in multilabeled graph databases. *17th IADIS International Conference Information Systems*, 2024.

[PVK17] Jaroslav Pokorný, Michal Valenta, and Jiří Kovačič. Integrity constraints in graph databases. *Procedia Computer Science*, 109:975–981, 2017.

[PVZ25] Jaroslav Pokorný, Michal Valenta, and Jiří Zikán. Validation of business process models using graph databases. *m*, 2025.

[RP17] Zdenek Rybola and Robert Pergl. Towards ontouml for software engineering: Transformation of kinds and subkinds into relational databases. *Computer Science and Information Systems*, 2017.

[RWE15] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, 2nd edition, 2015.

[Uni] Archived version of Uniqway website at https://web.archive.org/web/20221121120736/https://www.uniqway.cz/.

[Zik23] Jiří Zikán. Implementation of ontouml schemas in graph databases – case study. Master's thesis, Czech Technical University in Prague, 2023.